PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : H04L 29/06

A1

(11) International Publication Number: WO 00/56027

(43) International Publication Date: 21 September 2000 (21.09.00)

(21) International Application Number: PCT/CA00/00276

(22) International Filing Date: 15 March 2000 (15.03.00)

(30) Priority Data:
60/124,487 15 March 1999 (15.03.99) US

(71) Applicant: TEXAR SOFTWARE CORP. [CA/CA]; Suite 135, 1101 Prince of Wales Drive, Ottawa, Ontario K2C 3W7 (CA).

(72) Inventor: BACIC, Eugen; 56 Castlethorpe Crescent, Nepean, Ontario K2G 5R1 (CA).

(74) Agent: MITCHELL, Richard, J.; Marks & Clerk, P.O. Box 957, Station B, Ottawa, Ontario K1P 5S7 (CA).

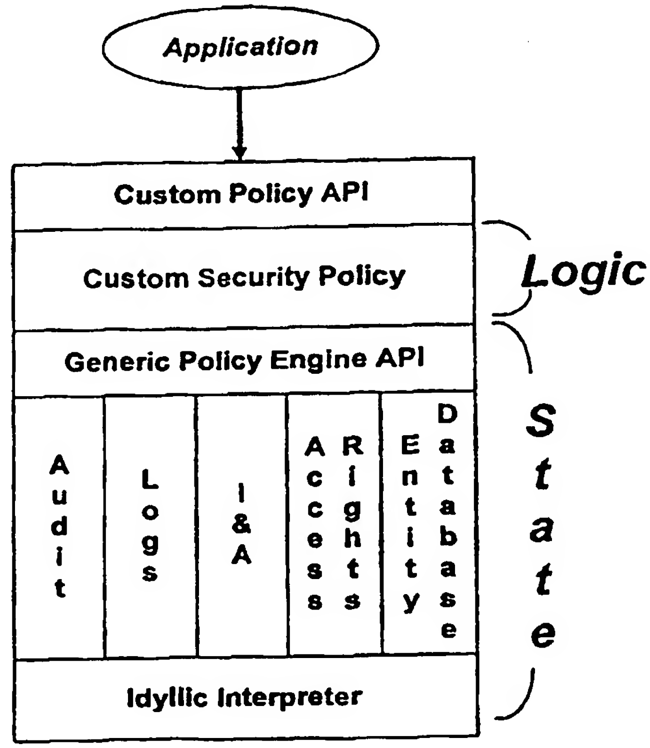
(81) Designated States: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: COMPUTER SECURITY SYSTEM

(57) Abstract

A generic policy engine (GPE) uses a verifiable, Scheme-like language to generate security policies ranging from the classical, hierarchical models to modern, commercial models. The GPE provides system designers with well-known security entry points, a generic definition of "object", and a means to manipulate these objects in terms of a security policy. The centralized nature of the system allows for experimentation with different security policies rapidly and economically.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

Computer Security System

Cross Reference to Related Applications

This invention claims the benefit under 35 USC 199(e) from US provisional application no. 60/124,487 filed on March 15, 1999.

Field of the Invention

This invention relates to the field of computer networks, and in particular to a security server for providing a network with sophisticated access controls.

Background of the Invention

As computer networks grow, security is becoming more of a concern with each passing day. Organizations view and relate to information differently and have differing requirements for the protection, dissemination, and modification of their information stores. Most organizations are moving towards heavily interconnected systems with links to the Internet. System architectures that were safe, due to limited accessibility, even a few short years ago are now being designed with wide access capabilities to meet the requirements of internal users, shareholders, customers, and clients.

For example, suppliers may wish to give third party customers limited access to their networks in order to facilitate design, ordering and accounting functions. Such third parties must not, however, have access to confidential corporate data, although in the case of co-operative design work, for example, there may be specific data files that the customer is authorized to access that would not normally be available to external organizations. This requires the ability to exercise highly sophisticated access control.

To meet security concerns, many organizations have opted for firewalls, virtual private networks (VPNs), and virus protection. These are commonly referred to as first generation security solutions. General access to host systems is provided based on the premise that once authenticated, users can be given full freedom to perform their duties.

Not only do first generation products protect only the perimeter, but they are islands of security, each performing a single task very well. They operate under their own control and their own rules; they do not play well with others, such as in the case of the third party customer example given above. These products have been defined and refined to control external access to information and ensure that only legitimate users gain access to

networks and their resources. Once a user is past these defences, little, if any, security exists to protect valuable corporate information assets.

Windows NT and UNIX systems, for example, offer access controls, but these are often implemented incorrectly, or disabled altogether, due to previous bad experiences on the part of users or systems administrators. Moreover, they are sufficiently different to offer little help in integrating security.

First generation companies cannot obtain the trust required from their competitors to create integrated solutions. The fear of competitive advantage ensures that players will not engage in an integration effort. While some companies have emerged to provide a limited form of integration for audit, they do not address the other concerns that security officers have, nor is it clear that they will provide solutions in the foreseeable future.

Reliance on first generation products that protect the periphery of the information rather than the information itself is no longer sufficient. Security requirements are changing, placing new demands on security officers and system administrators. They must now fulfil six security demands:

Integration among existing security products.

A high water mark for overall network security.

Controlled trust between systems even within an Intranet.

Centralized and uniform controls

Compartmentalized and configurable information controls.

Flexible and customer-oriented security rules.

First generation security products can be viewed as offering a wall of protection around information stores. Even though this wall may be sufficiently strong to thwart outside attack, it does not dissuade attack from the most common access point: the inside. These products are not designed to stop *authorized* users from accessing information. They are meant primarily to authenticate that a user is authorized to access the resources on the network. Once inside, few checks are performed, giving the user free rein.

Because of their distributed nature and the requirement to secure communications between machines, networked architectures are much more difficult to secure than single machines. This inherent difficulty is exacerbated by the fact that most networks do not

have security systems and policies which were designed for the entire network. In most cases, security has been grown by combining preexisting security systems as individual machines are connected.

With advances in network computing, increased requirements to share information and processing power among physically separate locales, and worries about information sensitivity, existing security can no longer suffice. When two or more computer systems are linked, their security policies often clash and overall security actually diminishes. If policies have evolved over time, rather than being designed for the network, solving these problems becomes more difficult.

To deal with combined legacy architectures, encryption is used to at least guarantee that information gets safely to its destination. Encryption, however, is a coarse technique for protecting information from disclosure. With no standardized method of creating security policies, encryption has become the de facto method of protecting information network-wide, even though its use surrenders the fine-grained control that was available prior to the networking of the computers.

The need for a security infrastructure has been fueled by the desire to provide new solutions in the face of increasing incidents of unauthorized access to and manipulation of computer systems, data, and communications. Malicious misuse of computer systems can be classified into three groups based on the origins of the threat:

1. **Outsider Threat** – This is an external individual or group attempting to breach the security of the system. Outsiders breach communications access controls but operate under the constraints of the communications protocols. This is the standard cracker attack. Outsider attacks are typically defended against by proper system administration and correctly designed and implemented access-control protocols and mechanisms, such as virtual private networks (VPNs) and firewalls.
2. **Malicious Software** – This is a piece of malicious code introduced into the system. The attack takes place within the communications perimeter, but remains bounded by the general access available to the operating system and the executing user. Malicious software may be introduced with or without a user's consent. The most common forms of this attack are the virus and Trojan horse.
3. **Insider Threat** – Here, the perpetrator is an individual with authorized access to the system. An insider may have wide-reaching control of the system or its components. The perpetrator may opt to replace hardware or software, and may observe any communications channel. This attack occurs within the boundaries of the VPNs or

firewalls as the perpetrator is an authenticated user. Insider threats commonly come from disgruntled or compromised employees.

A solid defense against insider threats can deter perpetrators by ensuring that they will get only a poor return on their investment. Such a defense can limit the damage done, minimize the information stolen or modified, and ensure that the perpetrator can be caught. In many cases, such a defence can stop most attacks, alerting authorities and ensuring that the threat is limited to system components which the insider generally has access to.

There is a need for a second generation security product that provides security across the network, security which is highly trustworthy, which is configurable to the needs of clients, which can be fully integrated with existing technology, and which is centralized for audit and administrative purposes.

Security policy refers to the rules governing the flow of information between two entities via predefined means and channels. A *policy engine* is a program that defines in logic those governing rules for a particular security policy, typically located within a trusted operating system. This patent presents a new method for the generation of policy engines using a high-level, verifiable language and generic security functionality.

Conceptually, security policies are accomplished in two stages, *analysis* followed by *synthesis*. In the first stage, a natural language description of the security policy model is created so that it can be verified as reflecting the policy for the product being developed. The second stage describes this security policy model as a contrivance of logic within the actual product. Thus security policies rely on the correct modeling of the policies of the target environment and the ability of the designers and implementers to translate these models into functioning security policies within the product.

Historically the development of security policies has tended towards meeting the perceived needs of the largest market and embedding a static security policy within the product, typically an operating system. The first such system was Multics which implemented the Bell-LaPadula Security Policy Model which closely reflected the requirements of the military establishment: the implementers. As time progressed and security became a serious concern to commercial interests, the security policies found

within existing operating systems did not adequately reflect their needs. Further aggravation was wrought by the fact that the embedded security policies could not be modified. In fact, the system was meant to handle information security in but one manner and if the methodology did not match that of the purchaser, the client was forced to either adjust their policies to more closely match that of the system or operate with less security than would be desired. To further aggravate the issue, even when two systems implement identical security policy models, say Bell-LaPadula, the implementations of these models within the products security policies don't match and communication between two machines often leads to security breaches.

Summary of the Invention

According to the present invention there is provided a method of controlling access to a network wherein security policies are determined by using a verifiable language consisting of formal definitions of the syntax and semantics.

A generic policy engine (GPE) uses the language to execute a policy in order to mediate access to an object entity by a subject entity. The invention abstracts the security information from the physical data.

Besides eliminating the requirement to custom develop security policies for every product under development, formal descriptions of security policies by means of a *language* allows the development of a semantics-directed security policy generator, one which simplifies formal proofs of the validity of the security policies and ensures valid generation of the security policies for inclusion within a product.

Several methods exist for formally describing the semantics of programming languages, and hence of security policies. Denotational Semantics has gained particularly wide acceptance among semanticists and language designers. This patent bases the verifiability of the language defined on the provability of systems developed using Denotational Semantics, such as VLISP.

In accordance with a further aspect the invention provides a security policy engine known as Generic Policy Engine (GPE) for controlling access to a network in accordance with policies determined with the aid of a verifiable language consisting of formal definitions of the syntax and semantics.

Security policies can be developed more efficiently, in a greater variety, and with comparable or greater trust than with traditional implementations. Security policies developed with the Generic Policy Engine (GPE) using the security language can be customized to reflect end-user security requirements as opposed to single-purpose security policies in common usage today. In order to provide for a generic policy engine a general, information neutral architecture must be used, one that is applicable across all operating systems or applications (henceforth collectively called "products"). Information must be encapsulated by the security measures required. The granularity of the information encapsulated defines the granularity of the security provided. The information thus encapsulated is the entity against which the GPE operates. By removing the restriction of specific products or data storage technologies, the GPE provides a trusted, extensible mechanism by which to implement security not only on a local system but also across heterogeneous networks. Providing custom security policies that provide equivalent trust to those currently in use and with similar efficiency would be a major advancement in the information security field. To attain this level of flexibility with the Generic Policy Engine one must have at one's disposal a high-level, verifiable language. All security relevant architecture must be written using this language which provides both the formalisms required for high security systems as well as the implementation specifics necessary to integrate it into a viable product. Such a language, which is referred to as Idyllic, will be described along with its use as the base language of the Generic Policy Engine.

Idyllic is a Scheme dialect. The Generic Policy Engine comprises a language interpreter plus the associated security extensions and functionality coupled with the ability to generate executable security policies. Thus, Idyllic provides a representation to easily define and implement verifiable security policies.

Entities are the unified security object against which all security related functions are performed. Both the base security application and any ancillary security relevant applications utilize entities in order to manage and maintain the security attributes of the applications. By adopting an entity-based approach to the name space being secured, the advantages of encapsulation and, to some extent, inheritance can be leveraged to create a succinct language definition for the creation of security policies. The new approach is to

utilize a language renown for its small footprint, efficiency, and simplicity and use it to implement security policies. Embedding the GPE as a secure extension with a system provides a trusted, extensible mechanism by which to implement security not only locally but across networks, within operating systems or applications, such as Web servers. The Generic Policy Engine provides system and application developers a means by which to provide equivalent trust to systems currently in use with smaller footprints and with at least equivalent verifiability, speed, and efficiency.

In accordance with the invention security policies can be developed more efficiently, in a greater variety, and with comparable or greater trust than traditional implementations. Security policies developed with the Generic Policy Engine can be customized to reflect end-user security requirements as opposed to single-purpose security policies in common usage today utilizing an information neutral architecture applicable across operating systems and applications. As proof of concept, a working Bell-LaPadula Security Policy model has been implemented as has a Message Trusted Guard.

Brief Description of the Drawings

The invention will now be described in more detail, by way of example only, with reference to the accompanying drawings, in which:-

Figure 1 is a Stylized Diagram of the Generic Policy Engine;
Figure 2 is an overview of the Generic Policy Engine;
Figure 3 shows a Classical Subject/Object Interaction as per TCSEC;
Figure 4 illustrates the Dual Requirement for Object Reuse at the System and GPE Level.

Figure 1 illustrates a system that produces security policies from formal definitions of the security policy model using a higher order, lambda calculus based language. This provides several advantages. First, it creates a uniform, universal language for developing and proving the correctness of security policies and their models. Instead of the current assortment of policy verification mechanisms developers would have one language, one semantics, and one syntax to implement their security policies. Second, the proofs for the security policies could be automatically generated from the formal semantics of the base lambda calculus. Third, by providing a uniform language environment, connectivity

between security policies can be improved since they would be based on higher level semantics than is currently available.

The GPE must provide a one-for-one mapping between the entities to be protected in the calling application and the security information maintained by it. In order to accomplish this, the GPE utilizes an entity-based approach defining an encapsulation with all the security relevant components within its boundary. Security relevant application programming interfaces (APIs) are provided so applications can manipulate security attributes and determine proper information flow, call the security policy, examine audit logs, and perform regular maintenance on the entity. The GPE's purpose is to provide sufficient security to an application so that the application requires only minor security specific instructions to meet even the highest security requirements of the various international evaluation criteria.

Figure 2 presents an overview of the Generic Policy Engine. The GPE is written in and controlled by Idyllic. The language is fully extensible and comes with a highly capable library of functions. The GPE's primary data structure is called the entity containing all the security data pertaining to an entity under its control along with the application programming interface (API) to manipulate the entity. Customizable elements, which can differ from application to application, can be provided by application programmers and are referenced via the APIs. The two programmer configurable components are the I&A mechanism and the security policy.

The various components of the GPE, which are discussed in more detail below, are the structure of the entity, its data elements, the facilities provided in the various APIs, the functionality provided by Idyllic, and the security functionality.

Mechanisms are put in place to uniquely identify and authenticate users and to dynamically track their actions within an application or system. Security controls, regardless of type, are not foolproof and to ensure a system of recourse after a security breach a non-circumventable, unalterable, continuous audit mechanism must be in place and operational. A base set of auditable events must exist which can never be disabled. The routines must be non-circumventable ensuring that neither accidental nor intentional modification of the audit system occurs.

The unique identification and authentication of every user or a particular user or system ensures that individuals utilizing the system can be held accountable for their actions. The mechanics behind identification and authentication can vary greatly and as such the core elements should be customizable to suit the threat assessment and the particular architecture. The guarantee of a minimal auditable event list is vital to maximize the trustworthiness of an application.

Information flow is controlled so that access to and manipulation of information is restricted to a specific set of individuals utilizing specific tools. As such there must be a mechanism provided to store and manipulate access controls for the various entities that are to be controlled. Each entity contains its own access controls entries to which programmers have access via the Access Controls API. The Access Controls data elements themselves are accessible only to the Access Controls API. The API is the sole method of retrieving pertinent information concerning an entity so that the security policy can operate effectively. The security policy itself is programmable by the developers though the Security Policy API is fixed. This provides the maximum flexibility: auditable events can be established at the API junction while removing any concern that the developers need remember to insert specific auditable events into the security policy they are writing. The security policy is customizable but is restricted to using the Access Controls API to access the Access Controls.

As a general purpose programming language additional security policies can be modeled which enhance, complement, or override the default security policy implemented within each entity. Security policies written in Idyllic can be modified to reflect the requirements of the applications developers thus reflecting the general policies and procedures pertaining to information flow and control for a specific organization. Default security policies, those applicable to all entities, can be provided to simplify security policy maintenance. Furthermore, compatible and cooperative security policies can be written to represent the nuances particular to a given department or section.

A virtual machine architecture for Idyllic and the Generic Policy Engine has been adopted to maximize portability and simplify the core and primitive elements of the language. Virtual Machines have been shown to be small and efficient and relatively simple to port.

The following table outlines the base entity within the Generic Policy Engine that represents the actual data stored by the calling application. The entity is subdivided into seven basic components: unique identifier, entity type, authentication information, security policy, audit history, privileges, and access controls.

Unique Identifier	Defined by the developer, usually reflective of the underlying application or operating system.
Entity Type	The type of entity. There are two distinct types. Those of type *group* are those which define a group of entities for a particular purpose. Those of type *entity* are for all other entities. A strict entity can contain multiple groups to which it belongs. A group contains a list of entities forming the particular group. The delineation assists the GPE in quickly traversing associations in order to simplify the security policy's goal of determining whether or not an information flow is to be permitted.
Authentication	String or data required for access (i.e., Password, private key, etc). This can contain more than one authentication attribute, each associated with a mechanism within the security policy.
Security Policy	The security policy for the entity. Each entity can have a different policy or can adopt a general one for the specific machine, domain, network, or institution to which it belongs.
Audit History	
Owner	A history of who "owns" this entity, most recent 1st, original owner (i.e., creator) last including timestamps.
Last Modified	A history of who last modified this entity, most recent 1st, original creator of file last including timestamps.
Last Access	A history of who last accessed this entity and by what means, most recent 1st, original creator of file last including timestamps.
Purge Rate	When events should be purged from the audit log.
Privileges	Historically these have been hierarchical in nature, but more modern security policies require a more flexible alternative.
Groups	To which groups or roles does this entity belong?
Associations	To which associations does this entity belong? Typically these are roles and the like.
Level	What "security level" does this entity have? This is security policy specific in that some require an individual level, others a range, and other none at all. In the cases where it can be within a range, the first element of the list can indicate the currently valid level and subsequent elements can either list the valid range values or provide a minimum and maximum level as per the security policy.

Categories	To which "categories" does this entity belong? This is security policy specific.
Caveats	What "caveats" does this entity have as restrictions? This is security policy specific.
Access Controls	What are the controls associated with this entity? The two most common are Access Control Lists (ACLs) and Roles.
Read	What other entities can request disclosure of information from this entity?
Write	What other entities can request to update the information in this entity?
Execute	What other entities can activate this entity?
Delete	What other entities can request that this entity destroy itself?
Copy	What other entities can request this entity create a copy itself?
Grant	What other entities have been granted temporary owner-based rights?

In order to map the entities managed by the Generic Policy Engine back to the entities under control of the calling application there must be some unique mechanism in place which can determine absolutely which entity corresponds to which application controlled data item. The calling application will always utilize the unique name when requesting or adjusting information stored and managed by the Generic Policy Engine.

To access some entities it may be necessary to provide appropriate authentication, i.e., entities that hold user authentication information or protect data items that have been password protected by the system or owner. To meet expanding authentication needs and retain compatibility with existing identification and authentication schemes the entity allows for a series of zero or more authentication strings to be stored by the entity and access by the Identification and Authentication API (I&A API).

As changes are applied to the item being protected the entity must track the changes, when and by whom they were performed, and the status of the attempt. This information is maintained by the Generic Policy Engine's audit facility. It allows each entity to maintain their own audit information.

Access controls are, in reality, split into two distinct portions: *privileges* and *access controls*. The former is maintenance information used to track groups, associations, levels, etc. to which this particular entity is a member; included in the list of maintenance information are all those entities which reference this entity. Access controls can be broken out into five standard "controls": read, write, execute, delete, and copy. Any

number of these can be utilized, in any fashion required to assist the security policy in performing its duties. It is irrelevant whether the security policy is access control based, capabilities based, role based, or otherwise based. The information required to properly execute such policies is provided within the entity, and especially within the *Privileges* component.

Groups, associations, levels, categories, and caveats provide a mechanism by which like entities can be grouped together. For example, all entities classified *Secret* would have the same Level. Definition of a group, say *Engineering*, could further subdivide the entities classified as *Secret* to those that are *Secret* and available only to someone in *Engineering*.

Each definition of a privilege for an entity further restricts access. Similarly, when an entity becomes active, its privileges define the domain to which it belongs and is utilized by the GPE to determine which passive entities can be accessed.

The other privileges provide a further refinement along traditional lines, namely categories and caveats.

The security policy, dependent and written for a particular application, utilizes the access controls and privileges to determine what is considered appropriate information access. A pointer to the security policy is provided within each entity allowing for individual policies per entity *or* for policies to follow the information as it is copied from one host or application to another. In this manner the information understands its policy and the validity of requests made by entities outside its original domain. Thus, applications protected by Generic Policy Engine built security components could move information securely between themselves knowing that not only the base data is transmitted during the flow of information between the applications but that the security information is passed along as well. This can be utilized to ensure that information to which one individual has access remains with that individual, even though the information was moved to a secondary system for whatever reason. Historically, once information left one trusted system with a given security policy to another, the new security policy determined access controls. In this scenario, if the new security policy was not disclosure driven, for example, it might permit accesses to the information that would be forbidden under the original security policy. By having the security policy move with the entity it requires a

conscious effort by the owner of the item to modify the security policy to the policy of the new host application along with all that it entails.

The flexibility the entity provides, regardless of overhead, to handle any security policy, individually or in tandem, is unique and revolutionary but does not come without cost.

The base language, known as *Idyllic*, of the Generic Policy Engine is based on two simple premises:

1. Manipulation of user information must be done efficiently and with minimal overhead.
2. The language must be *provably correct* in order for the GPE to attain the highest levels of trust within the various criteria.

Premise 1 removes most languages from contention, most notably Smalltalk and the other object oriented languages. The class library overhead of OOP languages is unnecessary baggage that in turn impacts on Premise 2.

Premise 2 requires a simple language that is easily understandable. In computer security, the development language for secure components must be *well defined*. To be *well defined* a language must have a formally defined grammar, a stable history (few, well defined and delineated changes over time), and supporting documentation available to the computer industry. Most computer languages (i.e., C/C++, Smalltalk) meet these requirements. However, for a *language* to be provably correct requires that all *aspects* of a language be properly defined and mathematically (or otherwise) provable to function as advertised. This would include all class libraries, control mechanisms, etc. Classical languages such as C/C++ and Smalltalk have large class libraries that preclude them from easily being proven correct.

The major differences between Scheme and Idyllic are that Idyllic has no provisions for:

1. lazy evaluation;
2. use of rational or floating point numbers; or
3. for calls with current continuations.

Strings are provided but at a drastically reduced functionality than found in Scheme. As programs written in Idyllic have no need for I/O, no method of I/O is provided in the

production environment, though one is provided via special debugging routines in the debugging environment.

Idyllic's control structures resemble Scheme.

The standard Boolean values for true and false are written as `#t` and `#f`, case is unimportant. The empty list, `()`, can be used as false, unlike some implementations of Scheme. In Idyllic, only `#f` or `()` are considered to represent false in a conditional expression. `#t` and all non-empty lists are considered to denote true. Note that the symbol `nil` is undefined in Idyllic.

(boolean? *obj*)	*boolean? returns #t if* obj *is #t, #f, or () and #f otherwise,*
(not *obj*)	*Not returns #t if* obj *is false, and #f otherwise*
(or <*test*$_1$> ...)	*The test expressions are evaluated from left to right, and the value of the first expression that evaluates to a non-false (not #f) value is returned. If all expressions evaluate to a false value, #f is returned*
(and <*test*$_1$> ...)	*The test expressions are evaluated from left to right, and the value of the first expression that evaluates to a false (#f) value is returned. If all values evaluate to non-false values, the last expression's result is returned*

A *predicate* is a procedure that always returns a boolean value (`#t` or `#f`). An *equivalence predicate* is the computational analogue of a mathematical equivalence relation (it is symmetric, reflexive, and transitive). Of the equivalence predicates found in Scheme only `equal?` is implemented in Idyllic. This is primarily for reasons of clarity rather than efficiency since `equal?` is usually the least efficient equivalence predicates available.

(equal? *obj*$_1$ *obj*$_2$)	*Equal? returns #t if* obj$_1$ *is equivalent to* obj$_2$.

Note that the advantage of using `equal?` as the only equivalence predicate is that it allows the policy designer to focus on the task at hand rather than on whether or not particular entities are equivalent. `Equal?`'s greatest advantage is that, generally, if two

expressions print the same they are equivalent. This is not true of most equivalence predicates available in Scheme.

Idyllic has but one type of number: integers. The reason for not implementing any other form of number is that, historically, operating systems and their components have never required floating point nor rational numbers, let alone complex numbers. To that end, Idyllic contains only integers.

An implementation of Idyllic must support integers throughout the range of numbers that may be used for indices of lists, vectors, and strings or that may result from computing the length of a list, vector, or string. The `length`, `vector-length`, and `string-length` procedures must return an integer. Mathematical functions always return an integer result.

Idyllic includes but a fraction of Scheme's control structures. Scheme allow for numerous looping constructs as well as numerous conditional expressions. Idyllic, on the other hand, has opted for simplicity. Programs executing under Idyllic need to be trusted and to be trusted they must be simple. Additional layers of complexity simply cloud the issues of security, complicate formal proofs, and obfuscate correctness.

To that end, Idyllic contains only eight control structures: `apply`, `cond`, `define`, `lambda`, `let`, `map`, `quote`, and `set!`.

(apply *fn largs*)	*Calls the function* fn *with the elements of the list* largs. *Returns the value* fn *typically returns after evaluation. For functions that accept* no *parameters,* largs *is the empty list.*
(cond *<clause>* ...)	*Each* <clause> *should be of the form* *(<test> <expression> ...)* *where* <test> *is any expression. The last* <clause> *may be an "else clause", which has the form:* *(else <expression> ...)* *A cond expression is evaluated by evaluating each the* <test> *of each* <clause> *until one evaluates to true. The associated list of* <expression>s *is then evaluated. The final* <expression>'s *value is returned by the cond. If no* <test> *expression evaluates to true, then the else clause is evaluated. If no else exists, #f is returned.*
(define *<var>* *<exp>*)	*Binds the expression* <exp> *to the variable* <var> *at the current level of scope. This is the only method by which a variable can be created into the current enviroment. Further, definition of a function is performed by having* <exp> *be a*

	valid λ (lambda) expression. Once bound, a function can be called by simply placing parenthesis around the variable along with any required parameters.
`(lambda <args> <body>)`	<args> *is a formal parameter list and* <body> *is a sequence of one or more expressions. The lambda expression evaluates to a function. This* lambda *is the only method by which programmable information can be stored within Idyllic. Evaluating the lambda expression along with the appropriate number of parameters is the method of calling functions in Idyllic. By using define one can store the definitions of the various functions within the current scope.*
`(let <bind> <body>)`	*The let expression is, in fact, syntactic sugar for a lambda. However, to aid in readability of Idyllic code, the let has been allowed to be the one exception that defines the rule of Idyllic's simplicity.* <Bind> *is a list of variable/expression pairs evaluated in turn with each expression being evaluated and bound to the provided variable. Once the binding is complete, the* <body> *is evaluated and the evaluation of the final expression found within the body returned.*
`(map fn list)`	List *must be a list and* fn *must be a function taking as many arguments as there are* lists. *If more than one* list *is provided, then all must be the same length. Map applies* fn *element-wise to all elements of the provided* lists *and returns a list of the results, in order.*
`(quote atom)`	*Evaluates the provided* atom *to create a literal constant defined as* atom. *Quote can be abbreviated as the single quote character "'".*
`(set! <var> <expr>)`	*The expression* <expr> *is evaluated and the resulting value is stored in the location to which the variable* <var> *is bound. The variable* <var> *must have been defined within the current scope or within the global scope. A variable is defined within a scope by means of the define structure, above.*

Security is concerned with limiting access between entities to only those authorized by the security policy. Security is usually defined in terms of an entity-based model which defines all entities controlled by the security policy as entities in one of two states: subject, the active state or object, the passive state, as shown in Figure 3. This definition has been shown to be limiting and causes problems when attempting to define security policies across an object-oriented system. The primary issue remains one of state and how the transitions from one state to another are accomplished.

In the present invention all subjects and objects are transformed into entities. By removing the distinctiveness of subjects and objects the definition of security policies becomes one of restricting flow between peers, rather than between "active" and

"passive" entities. An entity's passiveness or activeness is then directly related to whether the entity is actually being acted *upon* or performing the action.

An external application, such as an operating system, can utilize a policy written in the GPE. There are two customizable portions to the security policy: the I&A mechanisms and the Security Policy, both indicated in white. Login requests are handled via the standardized GPE API but the login mechanism can be any the developers feel would be most appropriate. This allows for rudimentary login facilities as found on most operating systems today or complex login facilities, such as those based on certificate authorities and X.500 directory services.

In Figure 2, a stylized structural of the Generic Policy Engine is presented. Each application wishing to utilize the Generic Policy Engine's security features would instantiate their own copy. Each GPE would be protected by the underlying hardware and memory management, ensuring information can't cross application boundaries. It is assumed that the GPE will be utilized in those environment which properly adopt memory management, object reuse, and 32 bit (or greater) architectures.

Without the adoption of these elements, the running application cannot be deemed trusted in the evaluation criteria sense of the word. Hence, the Generic Policy Engine would allow for multiple environments by providing for instantiations capable of creating their own secure environments and namespaces governed by the GPE for a particular domain such as an operating system or a database.

The utility of multiple environments is that various applications can use the Generic Policy Engine to create and manage their security policies and their particular authorization rules. This provides, in the case of an operating system, the ability to provide a *fully* functional security API for applications to use courtesy of the operating system and one that is as powerful as the native operating system's security system. Since all security is handled by the GPE, it can resist tampering and ensure logs are properly maintained.

Once authentication to the security policy has been achieved, all information access requests made to the Policy API are either boolean or string in value. Boolean values are returned when a request is made for specific access to a given controlled data item, such

as a file. Strings are returned when the request must return other than true or false, such as a cryptographic key, the true location of a file, current access controls for a particular controlled entity, etc.

It is impossible to have a security system without a mechanism of identification and authentication. Historically, identification and authentication (I&A) referred to the login/password pair which greeted users prior to the system allowing the user to begin processing. As time progressed, so have the I&A mechanisms until today we have numerous technologies ranging from login/password pairs to one-time passwords to crypto-tokens.

For the Generic Policy Engine to be generic from an I&A point of view it is sufficiently flexible to allow one or more varied identification and authentication mechanisms. To ensure that programming for the I&A mechanisms chosen remains as simple as possible, a fixed API is required guaranteeing programmers are met with a uniform interface regardless of underlying mechanism. To that end, the Generic Policy Engine must provide a standardized I&A API which interfaces to application specific identification and authentication mechanisms.

Further security is added by providing full audit trails that are customizable by the systems programmers. Since the security policy is available for perusal by the designers, the audit requirements of a particular application can be customized as required. As little or as much audit data can be written to various logs as required. The particular information written to the logs is also under the full control of the designers and no limits are placed on the complexities of the source defining how the audit records are created. This provides further flexibility in granting the security policy the option of calling helper functions to create detailed audit records in some instances while in others the records could be less detailed prior to pushing them out through the standardized audit mechanisms provided by Idyllic and the Generic Policy Engine. There exists, at all times, a lowest common denominator of auditable events that cannot be turned off. These are hard coded into the actual APIs ensuring that the system programmers cannot accidentally circumvent the audit system.

There are many forms of access controls. The primary form is known as confidentiality access controls, controls that define to whom information can be released. Conversely, the

control of who can manipulate information stored by an application is known as integrity access controls and operate along similar lines.

Access controls are further divided into the mechanism employed. The classic method is the use of access control lists that place the control information on each entity with explicit user names as to who can and cannot access the information. A modification of this form is role-based access controls whereby users are associated with roles and it is the role by which each entity determines whether or not an individual can have access. Other access control mechanisms operate similarly.

The Generic Policy Engine must be flexible enough to handle any form of access control, regardless of whether the controlling action is performed by the active entity, passive entity, or some combination thereof. The entity defines the information pertinent to the decision making of the security policy and contains sufficient information to model any known security policy. Since each entity controlled by the Generic Policy Engine is actually defined *by* an entity, the security policy defined is not restricted to a single form of access control. Thus, the application developer can define security policies that utilize information stored in an entity, regardless of state (active or passive) and determine via appropriate combination the access permissions.

For example, if the access control model is role-based then passive entities only indicate the type of access associated with a given role. The user entities actually define the roles to which they belong. Thus, if *User1*'s only role is *clerk* and *Entity1* only allows a *clerk* to read its contents, then *User1* will be granted read permission. If *User1* is removed from the system, *Entity1* does not need to be modified since it has no direct reference to *User1*. The same cannot be said for access control lists that reference the actual users in each entity's access list.

The entities under the control of the Generic Policy Engine encapsulators defining the security attributes of the data. The Generic Policy Engine ensures that information between entities is not accidentally shared. The Generic Policy Engine does provide object reuse so that entities no longer in use can be reclaimed. This call is part of the Generic Policy Engine's API.

The Generic Policy Engine has no mechanism to ensure that a trusted path has been set up between the application and the end user. This physical mechanism must be provided by the application whenever the user issues a predefined key sequence to the application. The security policy for the application can be coded such that it requires specific information prior to performing specific actions, such as information downgrades. This then becomes an issue addressed by the programmable security policy.

The actual code for the various parts of the GPE described above will now be presented, describing the entity data structure, its associated functions, and the resulting class-like structure.

In most cases, data structures are passive entities against which a system or application applies procedures or functions. In the Generic Policy Engine a single, state-aware data structure represents all the actual items within a system or application being secured, be they passive, such as files or active, such as processes. The entity can be in one of two states: active or passive. The first state is associated with information processing and typically operates against passive entities. To any active entity all other entities in a system appear passive. This allows for the manipulation and access of entities by entities to be generalized.

```
Entity      = STRUCTURE
                  UniqueIdentifier:  string
                  EntityType:        Atom
                  References:        LIST of Entity
                  Authentication:    string
                  SecurityPolicy:    Lambda, Default = NOACCESS
                  //* Audit Logs
                  Owner:       LIST of Entity
                  LastModified:      LIST of (Entity, Timestamp)
                  LastAccess:        LIST of (Entity, Timestamp)
                  PurgeRate:         Lambda, Default = NEVER
                  //* Privileges
                  Groups:            LIST of Entity
                  Associations:      LIST of Entity
                  Level:       LIST of Entity
```

```
Categories:        LIST of Entity
Caveat:            LIST of Entity
//* Access Controls
Read:              PAIR (Allow: LIST of Entity,
                         Deny:  LIST of Entity)
Write:       PAIR (Allow: LIST of Entity,
                         Deny:  LIST of Entity)
Execute:           PAIR (Allow: LIST of Entity,
                         Deny:  LIST of Entity)
Delete:            PAIR (Allow: LIST of Entity,
                         Deny:  LIST of Entity)
Copy:              PAIR (Allow: LIST of Entity,
                         Deny:  LIST of Entity)
ENDSTRUCTURE
```

Pseudo-code Data Structure for GPE Entity

Further, it is the entity, defined using pseudo-code above, itself that understands how it is to be manipulated or accessed and can control the type of access via the security policy embedded within it. This is accomplished by storing the security policy for each entity within the entity. This allows the security policy to travel with the entity, regardless of environment. Thus a user requesting access to a passive entity would have to be allowed access by the security policy of the entity itself, even though this policy may be a general one applicable to and shared by all entities in the system. By providing each entity with the ability to carry its own security policy the Generic Policy Engine provides a mechanism by which the information can be transmitted to another GPE controlled system, retaining the original access controls and security policy.

Idyllic implicitly manages all its data structures as symbolic expressions. The entity is by definition a symbolic expression and therefore properly handled by Idyllic. In order to differentiate between entities, each entity is uniquely identified so as to be accessible locally, internally to Idyllic, and externally to the calling application. Each entity has a current state, which indicates whether it refers to a user, process, or data. The data remains private with access permitted only via an exported API.

In a manner of speaking, an entity can be viewed as a database entry where the database management system is Idyllic and the database language is the GPE's APIs. The entity is managed as are all symbolic expressions within Idyllic. Accessing and manipulating entities via the APIs provides the security functionality associated with audit, identification and authentication (I&A), access controls, privileges, and security policy. The APIs can be subdivided into:

- General Entity API
- I&A API
- Audit API
- Access Controls API
- Privileges API
- Security Policy API

The API is used in order that the entity be useful outside the bounds of Idyllic. The entity is stored as a series of symbolic expressions and each is uniquely identified and mapable to an external calling application. This one-to-one relationship between the external representation of the entity and the GPE representation of the entity is vital to the security and integrity of the system. As each entity is manipulated it is fetched and stored in the access controls database. Each entity is *identical* in look and can be utilized for any purpose. This lack of typing is crucial to ensure that the GPE be sufficiently flexible in implementing any form of security policy utilizing its framework.

The "defaultSecurityPolicy" defined below is an explicit denial security policy. In other words, any action taken while this security policy was in force would be summarily denied. It must be changed, a conscious decision to override the default with a custom built security policy. It is in the interest of the application developer to utilize an appropriate and secure security policy.

```
(define (defaultSecurityPolicy . parameters)
   ;*
   ;* The default security policy declines every request for
   ;* information. This follows the standard edict of computer
```

```
;* security: if not explicitly allowed, it is denied. This
;* forces the developer to create a security policy. If the
;* developer opts to replace the "#f" with a "#t", it is a
;* conscience effort to forego creating a valid security
;* policy.
;*
#f)
```

Default Security Policy for the GPE

The choice of a monolithic entity encumbers the software by making it less modular. According to the current object-oriented programming style, a structurally cleaner and aesthetically more pleasing solution would be to have the various audit logs refer to an audit log, which in turn would contain the methods used to manipulate the logs. High-end security (e.g., the CTCPEC's T-5 and T-6 or the Orange Book's B3 and A1) require a tight coupling of all security relevant data within the GPE. In order to provide the tightest coupling, the GPE couples all the elements of the entity – the audit logs, the I&A, the access controls, privileges, and the security policy – directly to the data being controlled. This, in turn, results in all the security data elements being controlled and stored within a well-defined, controlled area, easily placed under the auspices of a memory manager or other well-defined hardware device.

In order to utilize a true object-oriented approach, the linkages between entities and the notion of what constitutes an entity would have to written into the underpinnings of *Idyllic.* Tracking and understanding the couplings for a given entity would require extensive syntactic extensions at the primitive level removing much of the simplicity and elegance found in *Idyllic.* This increase in complexity would not provide any additional security or trustworthiness to the GPE. Understanding the relevance of the links while utilizing memory management techniques to track and isolate the various memory elements that constitute an entity would be creating an artifice of memory management for the sole purpose of adhering to an object-oriented paradigm when one is not required. The use of a monolithic entity has been chosen to simplify and address the issues at hand, namely the creation of a security solution capable of generically describing and implementing security policies for any situation.

One question that does arise is whether a programmer has the flexibility to create additional audit logs or I&A mechanisms. Should the need arise, additional security logs or I&A mechanisms can be provided by utilizing existing mechanisms within the entity while ignoring other present capabilities. Capabilities not utilized can be viewed as being dormant, but available should the programmer wish to utilize them at a later time.

For each application requiring the functionality of the GPE a copy of the GPE is instantiated with its own memory area and entities. This adheres to the data separation requirements of many security criteria while simplifying the implementation. This also provides a clean and elegant solution to providing unique security policies to any application within a system requiring security functionality.

The entity with the elements expanded to include stylized contents within their association lists is shown below. Each entity is uniquely identified to the calling application by its *<uid>* (*u*nique *id*entifier) allowing for full cross-referencing. The cross-referencing is used to quickly remove all references to the entity should it ever be required. Hence, each time one entity refers to another, for whatever reason, the referenced entity is updated. Some fields contain lists that refer to various data instances such as time or entity identifiers (uid). For example, *owner* is a list of all the individuals who have ever owned a particular entity, with the first owner listed being the current owner and the last owner being the original owner, or creator. Similarly, *LastModified* lists in order, from most recent to least recent, which entities performed modifications to the entity.

Although it would be efficient in design terms to define the various audit logs as separate definitions pointing towards an *AuditLog*, it is more beneficial to define the logs within each entity so that the audit information can be properly protected. Access to the audit logs, elemental to the sanity of the entity and as a mechanism to provide additional warranties that the security has not been breached, requires access to the entity. Replication of I&A information and the possibility for error is removed, albeit at the expense of some processing overhead during audit log review.

Of particular interest in the above table is the use and reuse of the unique identifiers <uid>. Entities are utilized and referenced by the security policy, which places specific information within each. Thus, *Level* would contain pointers to such entities as *Secret*,

Unclassified, and *Top Secret*. The security policy would be able to request which the *Level* for a particular entity, compare it to other entities, and determine the validity of information access requests. Similarly, a group of entities can be placed into a particular role via the *Associations* tag. The <uid> for the entity could be *Programmer* and reference the entities which are programmers. And again, *Categories* can be used to define category subsets for use by the security policy. Thus a banking security policy could define valid categories as *Teller, Bank Manager, Branch Manager, Loan Manager, Clerk,* and *Financial Analyst* and compartmentalize information appropriately. This cross-referencing could be graphically illustrated to visually create representations of how information, one way or another, is related. This could greatly assist the security officer in determining whether there are any information flows that should not exist, such as data path from a higher hierarchical level to a lower one.

By providing a unified interpretation for *Entity* the GPE is generic. It is the interpretation of the information stored within the various elements of the *Entity* by the security policy that determines the form of the security policy, be it role based or access control based, or otherwise. Generic use of the entity generalizes security to its constituent components.

In any secure system it is vital that the security information remain protected at all times and that the mechanisms utilized to access the information remain immutable. Object oriented technologies provide a partial solution by formalizing the notion of data hiding with both the data and the procedures to manipulate the data stored within an object defined by a class. This notion can be extended to assist us in creating a viable storage mechanism for the Generic Policy Engine. Each entity corresponds to an item on the application side that the application programmer wishes to secure. The entity is the manifestation of the security attributes of the data in the application.

The combination of the data found within the entity and the procedures to manipulate the entity constitute the entity-based form of the GPE. This works well for static procedures which remain constant. The GPE must allow the flexibility to insert custom portions that reflect the unique identification and authentication mechanisms and the unique security policies of various organizations. At the same time, it must be able to put forward a well defined interface and functionality that would meet international security criteria. The solution is to utilize a standard API for each of the procedures while allowing specific

procedures, such as the Security Policy and Identification and Authentication routines, to have replaceable internals.

The following sections present the various procedures and their respective APIs. The final section presents the entire entity-based "class" for GPE-Entity that contains the data described above and all of the APIs and associated functions for manipulating the data within the confines of good security practice.

Creating a new entity requires no more than a call to *define-entity*. This special form accepts only a single parameter: the entity's name. It must be unique. *Define-entity* will ensure that the identifier provided is unique and if not, will issue an error. It is these structures that are maintained and manipulated by the GPE. They encapsulate the security of the items referred to by the calling application. Idyllic and the GPE maintain all the created entities in a persistent state. Each entity is a self-contained security module capable of determining whether or not another entity is granted access and the type of access allowed. Access to the entity is available only through the defined APIs.

```
    (define-entity <uniqueID>)
creates
    <uniqueID> <=
        (variables (<var> <value>)
                          :
                   (<var> <value>))
        (private-methods (message . args)
            ((message) <body>)
                          :
            ((message) <body>))
        (public-methods (message . args)
            ((message) <body>)
                          :
            ((message) <body>)))
```

General Structure of an Entity

The above code provides a completed entity structure for a generic entity, slightly stylized

for readability. As will be noted, many portions of the object-oriented paradigm are utilized in the design and implementation of the entity. However, it cannot truly be called object oriented as the only elements of object oriented computing utilized in the GPE are garbage collection (inherent in *Idyllic*), polymorphism, and encapsulation. Inheritance is notably absent. This is the primary reason why the GPE is referred to as entity-based.

```
    (define-entity GenericEntity)
creates
    GenericEntity <=
        ((variables (UniqueIdentifier  GenericEntity)
                    (EntityType        Entity)
                    (References        ())
                    (Authentication    (Active <Expiry> NoPassword))
                    (SecurityPolicy    defaultSecurityPolicy)
                    ;* Audit Logs
                    (Owner             (<creator>))
                    (LastModified      ((<timestamp> <uid>)))
                    (LastAccess        ((<timestamp> <uid>)))
                    (PurgeRate         (lambda () #f))
                    ;* Privileges
                    (Groups            ())
                    (Associations      ())
                    (Level       (<currentLevel> <min> <max>))
                    (Categories        (<currentCategory> <fullSet>))
                    (Caveats           (<currentCaveat>   <fullSet>))
                    ;* Access Controls
                    (Read              ((allow (GenericEntity
                                                <creator>))
                                        (deny  ())))
                    (Write       ((allow (GenericEntity
                                                 <creator>))
                                        (deny  ())))
                    (Execute           ((allow ())
                                        (deny  ())))
                    (Delete            ((allow (GenericEntity
                                                 <creator>))
                                        (deny  ())))
                    (Copy              ((allow (GenericEntity
                                                 <creator>))
                                        (deny  ())))
        ) ;* end variables
    ;*
    ;* Typical housekeeping functions (BetterHomes&Gardens)
```

```
;*
(private-methods (message . args)
   ((getReferences)
      <code>))
;*
;* The APIs, though their "C-like" form, look nothing
;* like what we have here. The APIs in reality are
;* syntactic sugar but use these calls. Pure GPE coding
;* would be too cumbersome otherwise.
;*
(public-methods (message . args)
   ((get)
      <code>
   )
   ((put)
      <code>
   )
   ((remove)
      <code>
) ) ;* end public-methods
) ;* end entity definition
```

Generic Definition of an Entity

UniqueIdentifiers refer to the <*uid*>s found within the GPE entity. This provides closure between the various classes, linking them together explicitly in a large lattice. Code fragments have been removed for clarity and brevity. <*creator*> is the creator of the new entity. Each entity is created by another entity, which is its creator. All entities can trace their heritage back to the original entity, the system entity, similar to the class Object in Smalltalk. The other parenthesized elements (< ... >) are self-explanatory.

A Trusted Computing Base (TCB) is the term that defines a boundary within which all controlled entities reside. In trusted systems, developers typically define which entities are under the control of the reference monitor and which are not. The defined subset of entities are considered to be *inside* the TCB boundary while the remainder are considered to reside *outside* the TCB boundary. The application using the GPE must provide the

additional security features of memory management and non-circumventability of the security policy.

By default the GPE provides a security policy. This security policy must be invoked by the calling application in order to properly function. There is no mechanism by which the GPE can guarantee the proper invocation, but by simplifying the mechanics behind the actual call, we can ensure that overhead and complexity are removed as viable answers to not utilizing the security procedures of the GPE.

Assuming the security policy is always invoked by the application, the application must be able to ensure the information being protected, say files within an operating system, cannot be accessed by standard operating system calls, such as low level disk reads. Although this latter form of access can be viewed as circumventing the security policy, many times this form of access is ignored especially when considering direct memory accesses. Modern memory management assists in preventing cross boundary access and alleviating many attempts at direct manipulation of memory.

In order for the GPE to be considered "trusted" it must meet the requirements which define a *reference monitor:*

- tamperproof;
- always be invoked; and
- small enough to be subjected to analysis and tests to ensure its correctness.

To be tamperproof the GPE must reside within a protected memory space. Modern computer hardware provides such mechanisms which are difficult to circumvent and meet the requirement of non-circumventability. Invocation requires that the application being protected *always* invoke the GPE for every transaction. This is something that falls to the application developer to address and if security is paramount, then invocation will occur. The final requirement is met by ensuring that both the GPE, Idyllic, and any security policies written in Idyllic are simple enough to be subjected to analysis and proven correct. As defined in the next section, policies that typically took thousands of lines in classical security policies can be defined in less than a hundred lines using the GPE.

One of the most difficult areas in computer security is ensuring that the system being developed is trustworthy. By using reference monitors and trusted computing bases, it is possible to increase the amount of trust that can placed in a specific trusted product. However, for each secure product developed, all aspects must be examined by an approved evaluation authority. This is a time consuming process. One of the main goals of the GPE is to reduce the amount of time required to evaluate a trusted product.

The controlled entities must be readily available between invocations. Idyllic can store all of the entities in its address space, readily available to the GPE. On large scale applications a caching scheme is used that is capable of ensuring entities are fetched and stored on disk depending on usage and when modified. Modifications to an entity must be written to disk immediately to minimize the likelihood that security information would be lost for whatever reason. The caching scheme would ensure that the most used entities remain in RAM, those less often used would be removed from memory after Idyllic guarantees that the information has been saved properly.

The simplest method of providing persistence is for the operating system to always keep the GPE and its associated entities in active memory. This would also provide for the quickest reaction time but may not be expedient when the number of entities controlled becomes large, as inevitably will be the case. Regardless of the system used to keep the entities readily available, it must ensure the isolation, availability, and correctness of all GPE data. Caching schemes are sufficiently well understood that one could be adopted for use by Idyllic to guarantee the integrity of its dynamic namespace.

Upon system shutdown, the GPE would have to be called to ensure orderly shut down of the security policy and proper storage of all databases. In a production version of the GPE a shutdown function would be used to initiate security shutdown. Similarly, when an application shuts down, it must properly terminate any copies of the GPE it instantiated.

Two versions of the Application Programming Interface exist: the internal, LISP-like API and the external, C-like API. The C-like API provides standard procedural prototypes for using the capabilities of the GPE. These C-like calls are converted into the internal LISP-like API. Provided in subsequent sections are the C-like APIs. The full LISP-like API is provided as part of the full definition of the Entity found discussed above.

Of note is that all LISP-like API calls are found within the entity as the exported public functions available to manipulate the data contained within. This gives the entity an object-based feel, retains strong encapsulation, and provides a single, cohesive unit to refer to the protected item in the calling application.

The GPE's purpose is to provide security to a calling application. Each application requests its own instantiation of the GPE. Each instantiation provides a fully functional copy of Idyllic and the GPE APIs. There are four function calls divided into two calls for opening a namespace of GPE entities for manipulation and a third to close the namespace. In order for the GPE to operate, a namespace must be opened and available to the calling application.

***gpeID = gpeCreate* ( )**	*Returns a reference to a new namespace. The application is expected to use* gpeID *whenever it requires access to the security functionality. If* gpeCreate () *could not create a new namespace, 0 is returned.* gpeID *is a long unsigned integer.*
	Once opened, all initialization to the global status of the GPE must be performed.
gpeOpen (gpeID)	*The application requests a previously opened GPE namespace to be opened by calling the function with* gpeID. *If the namespace exists, #T otherwise #F.*
	Once opened, all initialization to the global status of the GPE must be performed.
***gpeStart* ( )**	*All initialization and global sets must occur* prior *to the* gpeStart () *command being issued. This ensures that certain attributes are frozen for the running duration of the GPE, such as audit level.*
gpeClose (gpeID)	*Close the namespace associated with* gpeID. *#T on success, #F on failure.*

Instantiating a copy of the GPE

Every entity has a unique identifier provided by the calling application. This unique identifier (*uid*) uniquely identifies each entity managed by the GPE and provides a consistent mapping from the GPE entities back to the controlled items of the calling application. This simple, one-to-one mapping enhances the reliability and security of the GPE and ensures that identifier translation routines are not required thereby removing a source of possible error.

1.1.1 The Entity API

The GPE Entity API refers to what is normally called *class methods* in object-oriented programming. The functions defined in this API, see below, allow the programmer to manipulate the entity namespace in a coarse manner; strictly at the entity level.

gpeEntity (uid create)	*Creates a new entity within the namespace with the unique identifier provided. All information for the entity must be filled in piecemeal. Returns #T on successful creation, #F otherwise.*
gpeEntity (uid remove)	*Removes the entity referred to as uid. Returns #T on success, #F otherwise.*
gpeEntity (uid SP securityPolicy)	*Sets the security policy for the entity referred to as uid to securityPolicy. SecurityPolicy must be a valid lambda expression. Returns #T on success, #F otherwise.*
gpeEntity (uid SP DEFAULT)	*Sets the security policy back to the GPE default. Returns #T on success, #F otherwise.*

The Entity API

The GPE Identification and Authentication (I&A) API provides access to the I&A data of an entity. These calls allow for the creation of entities and the definition of unique identification and authentication of entities, regardless of whether they refer to a user or data object externally. The calling application need not track which users are valid nor which privileges they have. The GPE API provides system calls that can authenticate individuals. The GPE API provides additional system calls to add, remove, and update information concerning users of the application. The I&A database is maintained and protected by the GPE within each unique entity. The following table provides a summary of the Identification and Authentication API.

gpeAuth (uid Add authStr)	*Adds uid, plus its authentication string (authStr) to the entity referred to by uid. Returns #T on success, #F otherwise.*
gpeAuth (uid Remove)	*Removes uid (i.e., the entity it refers to). Returns #T on success, #F otherwise.*
gpeAuth (uid equal? authStr)	*Is the uid and authentication string (authStr) correct? #T if correct, #F otherwise.*
gpeAuth (uid Expires date)	*Sets the expiry date for uid. #T on success, #F on failure*

gpeAuth (uid Expires?)	*Returns #F if the uid's password never expires, the date of expiry otherwise.*
gpeAuth (uid SetPaswd old new)	*Sets the password to the new one provided the old password matches the existing one. Returns #T on success, #F on failure.*
gpeAuth (uid SetPassLen newlen)	*Sets the password length to the new one provided so long as it is between the predefined min and max password lengths. Returns #T on success, #F on failure.*
gpeAuth (uid deactivate)	*Deactivate the uid. #T on success, #F otherwise.*
gpeAuth (uid activate)	*(Re)activates the uid. Returns #T on success, otherwise #F.*

Identification and Authentication API

The Generic Policy Engine provides a built in audit capability capable of auditing every request by the calling application for mediation between a requesting entity and the targeted entity. Each entity, when accessed, logs all requests, the time of each request, and the status of the request to their internal logs. Each entity has a default audit level. These audit levels can be adjusted so more or less information is gathered. Therefore, to collect less information the audit level is lowered; to gather more, the audit level is raised. Entity level auditing cannot be completely shut off since a base number of auditable events *must* be tracked for security reasons, such as entity create, access, delete, etc. These typically correspond to system level actions such as file open, file close, file delete, add user, remove user, etc. Logs are lists with each entry stored as a sublist. Each entry contains the following information: requesting entity, time of request, action requested, and return status of request. This information is available, typically to the security officer and the audit tools, via the Audit API.

There exist four logs associated with each entity: *default log, owner log, last modified log,* and *last accessed log*. The *default log* is provided to allow calling application to insert specific information on events the application feels are security relevant. The *default log* can be used to create general logs by creating an entity whose entire purpose is restricted to logging information on behalf of the application. The *owner log* tracks who the owner of the entity is from the entity's inception through to its final destruction. The *last modified log* tracks all attempts at modification to a particular entity. And, the *last accessed log* tracks all accesses made to a particular entity.

MinimalAuditLevel	*The only auditing done is the bare minimum, namely recording entity creates and destroys.*
AuditLevel1	*Same as MinimalAudit but with the inclusion of open entity and close entity requests.*
AuditLevel2	*Same as Level1 but with the inclusion of modifications (write) to an entity.*
AuditLevel3	*Same as Level2 but with the inclusion of access (read) to an entity.*
AuditLevel4	*Same as Level3 but with the inclusion of copies so as to track if information is being moved from user to user via copies rather than standard reads.*
MaximumAuditLevel	*Same as Level4.*
DefaultAuditLevel	*Set to one of the above audit levels.*

Defined Audit levels In the GPE

For different applications, different levels of audit are required. Some require no more than the rudimentary audit granularity of when an entity was created and when it is modified. Others require a finer granularity capable of indicating whenever an entity has been accessed, or attempted to be accessed, by another entity. As the granularity becomes finer, the amount of information stored increases. This finer granularity provides a fuller picture of what is occurring from a security perspective, but requires a much larger storage allocation or more frequent examination of the logs. The GPE defines a specific set of audit levels, as defined above. These are adjustable prior to the issuance of the *gpeStart ()* command. Once the GPE starts, these global audit values are immutable.

gpeLog (GetLevel)	*Return the currently set level of audit.*
gpeLog (SetLevel <level>)	*The GPE provides predefined levels of granularity for audit. <level> must correspond to one of the predefined audit levels. Returns #T on success, #F on failure.*
gpeLog (setOptions ...)	*Sets the option list to the list provided (future).*
gpeLog (SetGMT)	*Sets the time to GMT, default for Time is local time.*
gpeLog (SetLocal)	*Sets the time to local time, this is the default.*
gpeLog (Time)	*Returns the current setting, either Local or GMT.*

Audit Log API (Global)

Manipulating the fixed elements is provided by a set of global audit functions, defined above. These are the functions which must be called prior to *gpeStart ( )*.

As time progresses in any security system, logs grow increasingly larger until they occupy all available space. There must be some mechanism to limit their size. In the GPE each entity has a lambda expression which defines the purge rate. This function determines when the log is to be purged, be it by age or by size. If by size, the log is truncated by removing the older data and retaining the newer information. If logs are never to be removed, then the purge date must be set as *never*. This ensures that the GPE never removes the logs but will, if the logs become full, halt the GPE.

Manipulating the logs requires an entirely different set of functions. Typically logs can be accessed in one of two ways: read and append. Many modern logs do not require the selective editing of the logs but rather only the wholesale purge, which is logged in the new log created after the existing log is closed for purging. This ensures a trace exists of all actions throughout the life of the system. The destructive functions are typically restricted to the *Security Officer* user.

The following API defines the common log access functions. The two system calls provided allow access to the four types of log, *defaultLog, Owner, LastModified,* and *LastAccessed*; <log> refers to one of these.

gpeLog (uid put <log> data)	*Appends data to <log> for the entity defined by* uid.
gpeLog (uid get <log> expr)	*Returns all entries for the entity* uid *which match* expr. expr can be a wildcard, which returns the entire log.

Audit Log API (Entity Specific)

Privileges in the GPE retain grouping information. There are two distinct types of group: hierarchical and non-hierarchical. Both are defined and accessible to security policies within the GPE framework and both are accessed using the calls defined in the API below.

All relevant grouping information is stored within the entity. Groups, associations, categories, and caveats are all non-hierarchical grouping mechanisms. They follow the common mechanisms found in many security policies which provide for the ability to group users or information in specific ways. For example, UNIX allows users to be grouped so as to provide specific groups with certain functionality. The military groups information via categories and caveats in order to limit who within a specific organization, regardless of clearance, can actually see the information. For example, information labeled *Secret NATO* would be visible only to those individuals holding a Secret clearance with the NATO category. Categories and caveats are often called *compartments*.

Hierarchical groupings are typically found in the military where the hierarchy is directly related to the classification system used to store information. The most common classifications are *Unclassified, Confidential, Secret,* and *Top Secret.* As a hierarchy, each is more restrictive than the former. Thus, Secret information is extremely sensitive but less so than Top Secret information. Typically, individuals with access to a higher level in the hierarchy also have access to information at lower levels in the hierarchy. Manipulation of the hierarchical mechanisms is performed in exactly the same manner as for non-hierarchical information.

gpePriv (uid put <priv> uid2)	*Adds uid2 to the <priv> list for uid. Returns #T on success, #F otherwise.*
gpePriv (uid remove <priv> uid2)	*Removes uid2 from uid's <priv> list. Returns #T on success, #F otherwise.*
gpePriv (uid get <priv>)	*Returns the complete <priv> list for the given uid.*

Privileges API

Privileges usually form the most secure mechanism of a secure system. As such, they must be easy to access. Although the entity's Privileges API provides sufficient mechanisms to extract the required information, there must be additional functionality provided in order to extract individual fields from within a record.

The granularity provided by the above functions is rather coarse. However, *Idyllic* provides a rich environment in which to extend the syntax by means of macros or user-defined functions. These extensions will be highlighted later when the GPE is illustrated by implementing a few well-understood security problems.

The access controls subsystem is actually divided into two distinct pieces: the data storing the information and the API providing the access routines. Since the entity stores all relevant information so as to be applicable to *any* security policy, the authors of a particular security policy must create appropriate helper functions that retrieve and manipulate the entity information. In the case of Bell-LaPadula, the helper functions would focus on retrieving classification levels for the mandatory controls. In conjunction with these, it is highly probable that the authors would provide functions to perform appropriate evaluations for set inclusion and dominance to further enhance readability. The GPE, however, provides sufficient syntactic enhancements to lessen the difficulty of programming in the GPE's base state.

gpeAC (uid put <right> allow uid2)	*Modifies uid's access controls to allow uid2 the specified <right>. Returns #T on success, #F otherwise.*
gpeAC (uid put <right> deny uid2)	*Modifies uid's access controls to deny uid2 the specified <right>. Returns #T on success, #F otherwise.*
gpeAC (uid remove <right> allow uid2)	*Removes uid2 privilege to access uid via <right>. Returns #T on success, #F otherwise.*
gpeAC (uid remove <right> deny uid2)	*Removes uid2 explicit denial of access to uid via <right>. Returns #T on success, #F otherwise.*
gpeAC (uid get <right> allow)	*Returns the allow list for the given <right> for the given uid.*
gpeAC (uid get <right> deny)	*Returns the deny list for the given <right> for the given uid*

Access Controls API

Access controls have a number of elements that must be easy to access. Although the entity's Access Controls API provides sufficient mechanisms to extract the required information, there must be additional functionality provided in order to extract individual fields from within a record. The above table defines the API for accessing and

manipulating the various aspects of the Access Controls data elements of the entity data structure.

The GPE provides for a generic API to access the security policy. This interface consists of a single call *gpe ()* defined in below. Valid actions are summarized in the subsequent table.

gpe (entity1 action entity2)	*Returns #T if entity1 can perform the indicated action against entity2, #F otherwise.*

GPE Security Policy API

Action	
read	*On access, read the information protected by the entity. On examine, view the list of entities allowed to read the information.*
write	*On access, write into the information protected by the entity. On examine, view the list of entities allowed to write information to the entity.*
execute	*On access, execute the information protected by the entity. On examine, view the list of entities allowed to execute the entity's information.*
copy	*On access, copy the information protected by the entity. On examine, view the list of entities allowed to copy the entity's information.*
delete	*On access, delete information protected by the entity. On examine, view the list of entities allowed to delete the entity's information.*
grant	*To which entities can this entity be "given", i.e., granted ownership.*

GPE API Commands

There must be two delineated access types: to the entity and to the data protected by the entity. In the former, access controls manipulation and examination is performed without accessing the data protected; examples from typical security includes changing the read and write permissions to a file or device. The latter represents actual access requests to the information protected by the entity. Although the Generic Policy Engine does not store the actual information, it acts as arbitrator on behalf of the calling application.

This delineation is vital since the Generic Policy Engine does not store the actual information being protected but merely the encapsulator and mediates information requests from the calling application.

The following table enumerates the valid calls to the GPE for invocation of the security policy. As can be seen, the invocation is a simple ***entity x action x entity*** triple which can map elegantly to any security policy. The simple invocation also allows for easy integration into procedural languages, such as C/C++ or Pascal.

gpe *(entity1 read entity2)*	*According to the security policy, can entity1 read entity2?*
gpe *(entity1 write entity2)*	*According to the security policy, can entity1 write entity2?*
gpe *(entity1 execute entity2)*	*According to the security policy, can entity1 execute entity2?*
gpe *(entity1 delete entity2)*	*According to the security policy, can entity1 delete entity2?*
gpe *(entity1 copy entity2)*	*According to the security policy, can entity1 copy entity2?*
gpe *(entity1 grant entity2)*	*According to the security policy, can entity1 grant ownership to entity2?*

Enumerated GPE Security Policy Function Calls

These calls work for any security policy that can be broken down into one entity requesting access to another through the provided actions (read, write, execute, delete, and copy). These requests for access to information can then be arbitrated by the Generic Policy Engine.

The security policy in the GPE is not integral to the interface but rather interprets incoming requests in a particular manner and passes these requests down to the meta Generic Policy Engine's Security Policy. The meta GPE Security Policy is the default security policy applied when an entity does not have a policy of its own. The customized portion, that dealing with *access*, is defined to handle the nuances of the particular security policy. Thus a GPE written Bell-LaPadula Security Policy handles mandatory access controls and discretionary access controls internally, accessing entity control information via the access controls API. Other security policies can also be defined to control access to the information by utilizing the generic nature of the GPE. The subsequent chapter illustrates the generality of the Generic Policy Engine by implementing a few security policies in common use today.

There are two distinct forms of security policy: the administrative security policy, discussed in this subsection, and the explicit security policy, discussed above.

The administrative security policy implicitly defines the behaviour of the explicit security policy by outlining exactly who, what, and how the various fields of an entity can be examined and updated. For example, Bell-LaPadula strictly restricts the motion of information. An entity at a given hierarchical level cannot be downgraded. A mechanism must exist to downgrade information for the system to be useful in the real world. This special case must be handled by defining individuals, known as security officers, with the special ability to circumvent the security policy. This power, obviously, must be controlled. In order to ensure the use is minimized, or at best properly controlled, *gpeManage ( )* contains special code which may circumvent the overall security policy. Any code that must circumvent the explicit security policy must be placed within *gpeManage ( )*.

gpeManage (entity1 field entity2)	*Returns #T if entity1 can update the indicated field of entity2, #F otherwise.*

GPE Security Policy Management Private API

The fields which are passed to *gpeManage ( )* are provided below.

UniqueIdentifier	Owner	Groups
EntityType	LastModified	Associations
References	LastAccess	Level
Authentication	PurgeRate	Categories
SecurityPolicy	AccessControls	Caveat

GPE Field List

Security operates under a simple assumption: every action must be approved. Once the action has been approved, the specific action can occur. Sometimes the action requires modifying one of the elements of an entity. In these cases, a subsequent call is made via

one of the previously defined APIs which perform the manipulation. All of the APIs, except for the Security Policy API, require only the targeted entity's *uid*. The GPE remembers the last access request, the requesting *uid*, the target *uid*, and the status of the request. For a subsequent call to one of the APIs the requester is therefore known. The action can proceed if and only if the status is *#T* and the API provided *uid* is identical to the one remembered by the security policy for the target. If not, the request is refused.

The following pseudo-code shows how the GPE is used in-line to check on the validity of one entity, in this case a user, attempting to access another entity, a file.

```
if gpe (user read file)
then
   readLine (file)
else
   warn 'access denied'
endif
```

The example illustrates a general use of the GPE to control access to a file by a user for a specific access method. If the security policy determines that the access should be allowed, the *readLine* is invoked; otherwise, a warning is issued. This general form is all that is required to invoke the security policy implemented in the GPE.

Although the security policy could be called for each invocation of the other API calls used in a program, calls are instead handled implicitly by a call to *gpeManager ()* by the API function.

As with any program that must maintain what is, in effect, a database, a number of housekeeping elements and routines are provided to keep the GPE operating smoothly and efficiently. Of particular note is the *References* list which provides pointers back to every entity which refers to a particular entity. This provides the GPE with the ability to rapidly remove references to a particular entity when it is destroyed and no longer required. Although the GPE could operate correctly without a *References* list, it would be inefficient to examine an entire namespace to determine which entities refer to a particular entity.

Object reuse comes in two parts: the reuse of the physical data storage used by a given entity protected by Idyllic; and, the actual security parameters stored for a particular

destroyed entity. The former case must be handled by the underlying application,.the latter is part of the garbage collection regimen of Idyllic. It must be guaranteed that the underlying application, prior to reallocation, clears any information stored within the entity. Similarly, the garbage collection routines for Idyllic must clear any security information from the entity prior to reallocation.

In some cases, such as UNIX, information at the memory level is cleared prior to reallocation. Disk files, however, must be cleared using an additional set of software either at file destruction time or just prior to reallocation. In the cases of other products, it may be necessary to create custom destruction routines to ensure the information is cleared prior to handoff to an underlying operating system or network. Encryption can be used to perform the object reuse either by the calling application or by Idyllic. It is logically infeasible to define interfaces to every possible kind of program to which the GPE may provide security.

In the Generic Policy Engine, the request to delete an entity automatically forces the system to delete the information associated with that particular entity. Prior to reallocation, Idyllic ensures any information within the entity is purged.

No trusted path mechanisms are provided by the Generic Policy Engine. A trusted path mechanism, if required for the evaluation of the application utilizing the GPE, must be provided for by the application itself since it is a direct user-system interaction. Additional logic in the security policy can be utilized to ensure the appropriateness of the trusted path command request.

Security requires that the security mechanism always be invoked. For any application wherein the flow of information from any point to any other point is well known and defined the GPE will work perfectly. Those applications where there is little clear definition as to where the information is supposed to flow and the paths taken, the GPE will be difficult to incorporate. That said, the latter will also be the least likely applications to be secured and even less likely to be evaluated at anything other than the lowest level of trust within any of the evaluation criteria currently available.

Thus, the GPE is based on the premise that it will be embedded within a properly modular system with clearly defined and controlled flows of information against which policies

implemented in the GPE can operate. Idyllic and its associated Generic Policy Engines do not rely on any specific operating system or application type. The only requirement is that the underlying operating system or application provide information to the implemented security policy in a format compatible with the GPE API.

The Generic Policy Engine provides core functionality for implementing security policies. It is not concerned with the details of each and every security policy but rather provides the basic elements required for any security policy. To that end, the functionality provided is simple, and elegant. Since Idyllic is a lambda-based language, extending the syntax is a straightforward exercise; syntactic extensions are used to address the readability limitations of the Generic Policy Engine's base functionality. The following paragraphs present a number of security policies which, in order to make the security policies more readable, are provided with their own system calls.

Bell and LaPadula defined their security policy in terms of subjects (active entities) and objects (passive entities). In the original definition, subjects were processes and objects were files associated with an operating system. For our purposes we shall define the Bell-LaPadula Security Policy in terms of their classical use where subjects define active users and objects define passive files and peripherals, such as tape drives and printers.

The Bell-LaPadula Security Policy is described in terms of a simple triple containing the user (or *subject*), the data (*object*), and the action. The two governing rules of the Bell-LaPadula Security Policy below.

Simple Security Property:	Also known as the no read up (NRU) rule, states that a subject with security label L_S can only read information of an object with security label L_O if and only if L_S dominates (is greater than) L_O.
****-Property:***	Also known as the no write down (NWD) rule, states that a subject with security label L_S can only write information to an object with security label L_O if and only if L_O dominates L_S.

Bell-LaPadula Security Properties

Each entity controlled by the Bell-LaPadula Security Policy retains state.

Implementing Bell-LaPadula in the Generic Policy Engine is straightforward. Subjects become active entities while objects become passive entities. In actuality, subjects would be defined strictly as processes in the calling application though with this implementation of Bell-LaPadula it does not necessarily have to be so.

There are many aspects of the policy that presuppose specific attributes existing prior to the security policy coming into force. Three of the most crucial are identification and authentication (I&A), trusted path, and audit mechanisms. These are typically deemed outside the realm of the security policy. The Generic Policy Engine, however, provides a unified approach to security and hence an integrated solution to I&A and audit. These will be utilized in the implementation to provide necessary identification and accountability information necessary for a properly defined security solution. Trusted path is attainable only by the calling application and cannot be completely addressed by the Generic Policy Engine, it is not implemented below. Object reuse requires corresponding code in both the calling application and the GPE. Within the GPE, object reuse is implicitly handled by housekeeping functions such as garbage collection.

The Bell-LaPadula Security Policy is known as an hierarchical model. This means that the information is segregated into distinct, hierarchical levels each separate and distinguishable from all others and each defined via a dominance relationship with the others. Typically the hierarchy consists of the military classifications *Top Secret*, *Secret*, *Confidential*, and *Unclassified*. They are related in that each in turn dominates the next in a simple mathematical relationship:

Top Secret > *Secret* > *Confidential* > *Unclassified.*

This allows for simple arithmetic rules to be utilized to determine whether an active entity can view an entity by what Bell and LaPadula called *dominance relationships.* For example, an entity at Top Secret would be invisible and unavailable to an active entity at Secret; conversely, an entity at Confidential would be visible and available to the same active entity.

But the Bell-LaPadula Security Policy has other aspects, namely categories and caveats that are *not* hierarchical in nature. A given entity is not bound to a single category or caveat as they are to a single level of a hierarchy. Categories and Caveats are viewed as sets with any entity within the Bell-LaPadula Security Policy belonging to a predefined

set. These categories and caveats are, typically, available to each level and to all entities though not all categories or caveats may be utilized for all entities. Namely, each entity is allowed a specific, predefined subset of the category and caveat sets. Each entity is then restricted to any and all subsets of that predefined subset, including the null set.

Bell-LaPadula models an active set of processes, one where information is moved from one secure or trusted state to another. In order for information to be moved between states it must first be created. If it is to be created, there may come a time when it needs to be destroyed. The Generic Policy Engine provides mechanisms to create and destroy entities and their associated linkages and information. For this implementation of the Bell-LaPadula Security Policy we require two functions as defined below.

createEntity *(type entity classification)*	*Creates one of the two types of entity required by the Bell-LaPadula Security Policy (subject or object). The classification level for the new entity is passed as a parameter. On successful creation #T is returned, #F otherwise.*
destroyEntity *(entity)*	*Destroys the entity and removes all references within the GPE namespace to it. Returns #T on successful destruction, #F otherwise.*

These two functions simply call the existing internal entity creation and destruction routines. All housekeeping functions are handled implicitly by the Generic Policy Engine leaving the security policy uncluttered by such matters.

The least privilege requires that the person performing a duty must be granted only those system privileges necessary to properly perform a particular duty. This is contrary to many systems, such as UNIX, that provide all or nothing privileges under the guise of a super user or administrator. The typical solution is to create privileges that ensure a particular task requires a particular privilege. Therefore, the tasks associated with an operator – mounting tapes, starting and stopping the printer queues, routine non-security related maintenance – would require *operator privilege*; the creation or modification of user accounts, upgrade or removal of applications, etc. would require *system privilege*. The privileges are typically associated with *roles* such as *operator* or *system administrator*. Applying this solution to UNIX would result in at least three distinct user

types: unprivileged user, operator, and system administrator. Security officer is typically added and is responsible for the system logs, maintenance of the security policy, security policy enforcement, and information flow, typically in the form of document downgrades.

Each entity must be assigned a privilege set at creation time. The privilege set is used to determine which functionality the entity can utilize during its operation as an active entity. Least privilege further stipulates that the entity activate only those privileges required to perform its duties.

The Bell-LaPadula policy does not require the inclusion of privileges, relying instead on the division of "subjects" and "objects" according to the defined hierarchy. By defining system applications as "System Low", i.e., below all other levels of the hierarchy, sensitive files can be protected from inadvertent modification. By defining unique categories specific to the various system administrative duties, specific tasks can be delegated to specific individuals. In this way we can subdivide all users into one of two types: privileged users and unprivileged users. We can further subdivide each of these groupings as required by allocating specific categories to differentiate specific privileged sets. For example, categories could be defined to represent the system administrator, operator, and security officer.

Mandatory access controls (MAC) provides system administrators the ability to control access to entities by active entities in a mandatory (system defined) way. These controls are regarded by some as system administrator discretionary controls over the entire system. MAC controls typically compartmentalize information. The most common use is to impose hierarchies upon the information such as unclassified, confidential, secret, and top secret. The rules governing MAC define groupings of information to which a active entity *must* belong *prior* to access. The GPE provides for *Levels* which are used to compartmentalize entities to better control information flow through the system. The compartmented entities are usually further subdivided by means of categories and caveats. The definition of valid sensitivity levels, categories, and caveats must be defined prior to the first utilization of the GPE. This is accomplished by defining each valid level, the explicit hierarchy they represent, and the valid category and caveat sets.

Using syntactic extensions that create and populates appropriate entities for us, we first create each hierarchical level.

```
(createLevel 'SystemHigh)
(createLevel 'TopSecret)
(createLevel 'Secret)
(createLevel 'Confidential)
(createLevel 'Unclassified)
(createLevel 'SystemLow)        ;* where the system files live
```

Once all the levels have been created we create the explicit hierarchy. Idyllic allows us to create the hierarchy simply as a list. This list explicitly defines the hierarchy and allows for simple arithmetic operations to be utilized to determine which security level dominates another relative to position in the list; the higher the position, the higher the level.

```
(define BLPHierarchy   (LIST SystemHigh
                             TopSecret
                             Secret
                             Confidential
                             Unclassified
                             SystemLow))
```

Categories and caveats are unordered sets and are similarly created. Instead of creating a hierarchy, they represent valid sets.

```
(createCategories  'NATO)
(createCategories  'NUCLEAR)
(define BLPCategories    (LIST NUCLEAR NATO))
(createCaveats     'CanadianEyesOnly)
(createCaveats          'PrivyCouncilOnly)
(define BLPCaveats       (LIST CanadianEyesOnly PrivyCouncilOnly))
```

Every entity begins with the following set as the default when created:

```
((SystemLow SystemLow SystemLow)    ([]     [])   // Categories
                                     ([]     []))  // Caveats
   Current     Min       Max        Current Fullset
         Levels
```

This triple defines the entity at *SystemLow*, and has empty sets for both the category and

caveat lists. One of the first actions performed is to upgrade the entity to its appropriate current, minimum, and maximum levels, which in our example would be anything from *Unclassified* through *TopSecret*; the GPE would also insert the appropriate caveats and categories for the current active set and the full set.

Therefore, an entity with a security level of:

```
((Confidential Confidential Secret) ([] []) ([] []))
```

is said to dominate one with the following level:

```
((Unclassified Unclassified Confidential) ([] []) ([] []))
```

since the current hierarchical level of the former is *Confidential* and that of the latter *Unclassified*, regardless of how high or low any given entity may reside.

Two system calls are typically required to utilize MAC: *setMAC ()* and *getMAC ()*. The MAC settings are usually defined as ranges:

$$\text{SecurityLevel} = [\text{min}, \text{max}]$$

$$\text{Categories} = [ct_1, ct_2, \ldots, ct_N]$$

$$\text{Caveats} = [cv_1, cv_2, \ldots, cv_N]$$

***setMAC** (entity classification)*	*Sets the security level, categories, and caveats for entity to the classification information provided.*
***getMAC** (entity)*	*Returns the classification information for entity as a triple. If entity doesn't exist, the request is refused.*

Discretionary Access Controls (DAC) correspond to Axiom 3 of the Bell-LaPadula model. DAC allows system users to restrict access of the entities that they own to specific active entities, in a discretionary way. The TCSEC requires at C2 and above that "access controls ... be capable of including or excluding access to the granularity of a single user" Some operating systems, most notably UNIX, do not support such permissions via self/group/public permission bits. True access controls follow those outlined in Multics and implemented by the Generic Policy Engine.

Permissions come in two primary flavours: role based and access control based. Both are very similar with role based access control defining what specific roles can or cannot access particular information and having all users within the system defined by specific roles. Strict access controls define specific users or groups of users as having access to particular information. Both are equally valid. For our purposes, we will use access controls since they more closely correspond to a majority of the Bell-LaPadula interpretations currently in use throughout computer security. Permissions are typically defined by read, write, and execute controls.

For our purposes we will define access controls as lists of the following form:

```
(deny  (. . .))
(allow (. . .))
```

for each of read, write, and execute controls. The full access control list (ACL) for any given entity controlled via the Bell-LaPadula Security Policy would have the following general format:

```
(read    (deny  (. . .))
         (allow (. . .)))
(write   (deny  (. . .))
         (allow (. . .)))
(execute (deny  (. . .))
         (allow (. . .)))
```

Therefore, an ACL for a given entity of the form:

```
(read    (deny  ( ))
         (allow (engineering eugen)))
(write   (deny  (engineering))
         (allow (eugen)))
(execute (deny  ( ))
         (allow ( )))
```

would permit engineering and *eugen* to read the contents of the entity while denying all others. Similarly, although engineering is explicitly denied write access, eugen is explicitly allowed access. If *eugen* was also a member of engineering the specific granting of a privilege *always* supercedes the general denial, and vice versa. In Windows NT this is

not the case. In Windows NT a specific grant of access can be superceded by a more general denial of access; this contradicts the intent of security policies based on Bell-LaPadula wherein the most specific grant of access always supercedes the more general denial of access.

This general ACL structure allows us to create any variation of permissions desired for any application, be it an operating system or otherwise. This differs from the mechanisms typically employed by trusted UNIX implementations, such as Trusted Xenix and HP-UX in that they employ a fixed size ACL (1 kilobyte per file) and utilize a triple (user, group, access permissions).

In order to manipulate the access control lists a number of system calls are required. The following table summarizes the three functions which provide the Bell-LaPadula Security Policy with the functionality required to manage the ACLs. Although not explicitly mentioned, the Bell-LaPadula Security Policy model requires mechanisms by which the ACLs can be updated, removed, and fetched.

***removeACL** (entity access privilege entity2)*	*Access can be one of deny or allow. Entity2 is removed from the provided privilege in the ACL.*
***modifyACL** (entity ACL)*	*Modify an existing ACL by replacing the ACL with the one provided.*
***getACL** (entity)*	*Retrieve the access control lists for entity. If entity doesn't exist, return #F otherwise the ACL.*

As entities are dynamically created and destroyed, the issue of object reuse must be addressed. Object reuse provides for the return of information to the system's free pool allowing the space occupied to be reused by other applications. Any system that uses Idyllic to implement and maintain its security policy divides the issue of object reuse into two distinct types: Idyllic object reuse and system object reuse. Figure 4 illustrates what happens when there is a request for the deletion or creation of a particular data file within the system. When a request is received by the system for the destruction of a particular data file, the system must ensure that the requesting entity has appropriate access. If all controls are correct, then the entity in question can be destroyed. This destruction is twofold: First, the instance representing the entity within the security policy under Idyllic must remove the entity and all references to it. Once that process has successfully

completed, the system itself can remove the physical storage for the entity and, if need be, sanitize it.

All security criteria require some form of object reuse that ensures that any reallocated entity's contents are independent from its previous instance. Some operating systems, such as UNIX, clear all memory prior to reallocation; others, such as DOS, do not. Therefore, depending on the operating system or application using Idyllic, it may be necessary to create an object reuse function to properly sanitize entities returned to the free memory pool.

In order for the Bell-LaPadula policy to operate a mechanism of identification and authentication must be in place. The GPE provides identification and authentication functionality fully capable of delivering sufficient uniqueness to describe all operational active entities within the model. For the purposes of this example a simple login ID and password suffices to properly and uniquely identify each active entity to the security policy. In order to facilitate the usage, the following routines are provided which provide I&A mechanisms for the Bell-LaPadula Security Policy.

Login *(user password level category caveat)*	*Attempts to log in the user with the provided password, level, category, and caveat. On success, #T is returned, otherwise #F.*
Logout *(user)*	*Logs the user out.*

Many high security systems provide a trusted path mechanism whereby a secured line of communication is set up between the user and the system. As the trusted path attention mechanism is typically hardware oriented, it is outside the Generic Policy Engine. However, the Generic Policy Engine does provide a system call that can be utilized to invoke specific system commands securely by the user.

The Generic Policy Engine implicitly handles the most common forms of audit found in many trusted systems. Auditable events such as entity creation, entity destruction, modification of an entity's security parameters, etc. are all logged by the Generic Policy Engine. Bell-LaPadula does not explicitly require audit since in a perfectly functioning Bell-LaPadula Security Policy, no breaches would be possible. It is only when the

implementation moves from the realm of mathematics to one of computer logic that errors can occur that cannot be easily verified. These errors could result in breaches of security with a resulting improper disclosure or manipulation of protected information. Audit facilities are therefore utilized in an attempt to catch breaches as they occur for various reasons including possible prosecution. However, for our purposes and since the Bell-LaPadula Security Policy does not explicitly require audit, it is sufficient to rely on what the Generic Policy Engine provides as default – namely, all major entity modifications are logged by the GPE for future perusal by appropriate security personnel.

Now that the entire infrastructure is in place, we need to define the actual Bell-LaPadula Security Policy. The implementation of the Bell-LaPadula Security Policy is defined below. Utilizing the extensions defined, we have avoided using raw GPE calls. This allows for a shorter and more readable Bell-LaPadula Security Policy implementation without diminishing the verifiability provided by the GPE security framework. When writing code with the GPE framework, it is important to remember the primary goals of the GPE, and, in fact, all security policy implementations: small, readable code that is verifiable and efficient. The following definition epitomizes this goal.

```
(define (Bell-LaPadula subject access object)
   (let* (
      (Subject-Level (getLevel      subject))         ;;* MAC info
      (Subject-Cats  (getCategories subject))
      (Subject-Cavs  (getCaveats    subject))
      (Object-Level  (getLevel      object))
      (Object-Cats   (getCategories object))
      (Object-Cavs   (getCaveats    object))
      (Allowed       (getRight object access 'allow)) ;;* DAC info
      (Denied        (getRight object access 'deny)))
;  in
      (and (or (= Subject-Level Object-Level)         ;;* MAC
               ;;* no read up
               (and (> Subject-Level Object-Level)
                    (eq? access 'Read)
                    (subset? Object-Cavs Subject-Cavs))
               ;;* no write down
               (and (< Subject-Level Object-Level)
```

```
                    (eq? access 'Write)
                    (subset? Object-Cavs Subject-Cavs)))

            (and (member subject Allowed)              ;;* DAC
                 (not (member subject Denied)))

   ) ) )
```

The simplicity of the above is a direct outgrowth of:

- the ease with which one can manipulate symbols in a symbolic language; and,
- the underlying functionality provided by the GPE which provides additional security relevant facilities. These additional facilities provide for the needed security functionality typically lacking in operating systems.

Standard access control lists are not typically defined within the two properties associated with the Bell-LaPadula policy even though they are always included within operational implementations of the security policy. For completeness, the above policy definition includes access control lists as defined on many systems.

Now, in the implemented prototype the Bell-LaPadula Security Policy uses readable names to differentiate various entities. If the GPE were connected to a working operating system, however, unique identifiers understood by the operating system would be used instead. The GPE requires only that the identifiers be unique, regardless of syntax. Most operating systems use numeric representations to ensure uniqueness. This is sufficient for the GPE.

With the Generic Policy Engine, the entire Bell-LaPadula Security Policy, typically thousands of lines of system code, has been implemented in less than 10 lines of code. This security policy would be embedded as the default security policy for all entities that were to be governed by the Bell-LaPadula Security Policy.

Even though there exists within the GPE and the earlier syntactic sugar routines for accessing the various security aspects of an entity, the security policy proper requires specific routines which return numeric values for the security information to be able to properly execute the security policy.

The Bell-LaPadula Security Policy definition utilizes five functions, as defined below, to assist in manipulating the security information properly and efficiently for the Bell-LaPadula Security Policy.

getLevel *(entity)*	*Returns the current security level for the entity as a numeric quantity.*
getRight *(entity privilege)*	*Returns the requested privilege as a list for the entity.*
getCategories *(entity)*	*Returns the current category set for the entity.*
getCaveat *(entity)*	*Returns the current caveat set for the entity.*
subset? *(set1 set2)*	*Returns #T if the set1 is a true subset of set2, #F otherwise.*

The Bell-LaPadula Security Policy controls the disclosure of information. To perform this task in a dynamic environment requires that the access controls be constantly updated with current and relevant information pertaining to valid subject-object (active entity-passive entity) interactions. The following figures illustrate how the access controls can be updated and how the Bell-LaPadula Security Policy is invoked to mediate subject-object interaction.

gpeAC (ebacic put write allow GPE.doc)	Request that ebacic be allowed to write to GPE.doc
gpeAC (jagoda put read allow GPE.doc)	Request that jagoda be allowed to read GPE.doc
gpeAC (ariana put write deny GPE.doc)	Request that ariana be denied write access to GPE.doc
gpeAC (goran put delete deny GPE.doc)	Request that goran be denied delete privilege to GPE.doc

Updating the Access Controls via the GPE

gpeAC (GPE.doc get read allow)	Returns all entities that are allowed to read GPE.doc

gpe (DOR get execute deny)	Returns all entities that are allowed to execute DOR

Examining Access Controls via the GPE

gpe (ed read GPE.doc)	returns whether ed can read the file GPE.doc
gpe (john write bp.doc)	returns whether john can write to the file bp.doc
gpe (dan delete john)	returns whether dan has the right to delete john

Access Control Mediation by the GPE

Message Trusted Guards (MTGs) are defined in terms of users and electronic mail addresses. Controls are placed upon the destinations to which specific users can transmit electronic mail messages. No restrictions are placed on incoming e-mail, even though a Message Trusted Guard is capable of controlling both incoming and outgoing connections.

Modern implementations of Message Trusted Guards are embedded within Firewalls, however, earlier this decade stand-alone MTGs were quite common and typically were thousands of lines of C code, much of which handled housekeeping functions for the security controls. The GPE implementation, described below, is less than 50 lines and offers the same functionality.

Implementing a MTG Security Policy in the GPE is straightforward with the active entity being the actual e-mail message/sender pair and the passive entity being the recipient/e-mail address pair. The action is always "send". Therefore the calling sequence to the GPE coded security policy is: *gpe (sender send recipient).*

There are two types of entities that are maintained by the Message Trusted Guard: senders and recipients. The most common implementation places restrictions on who can transmit to a given recipient. If a given sender is not explicitly permitted, transmission is denied. In order for the e-mail messages to be transmitted from the current site to a desired destination both the sender and recipient must exist. To exist the entities must be created. There may come a time when an entity needs to be destroyed. We'll utilize the Generic

Policy Engine's mechanisms to create and destroy entities and their associated linkages and information. For the Message Trusted Guard we will define two functions.

createEntity (type uid)	*Creates one of the two types of entity (site or user) with the provided unique identifier. On successful creation #T is returned, #F otherwise.*
destroyEntity (uid)	*Destroys the entity and removes all references. Returns #T on successful destruction, #F otherwise.*

These two functions simply call the existing internal entity creation and destruction routines. All housekeeping functions are handled implicitly by the Generic Policy Engine leaving the security policy uncluttered by such matters.

All users of a network capable of transmitting e-mail would be registered with the Message Trusted Guard when their accounts and e-mail are initially set up. Recipient restrictions are created by the security officer on an address by address basis. If a given address is not in the Message Trusted Guard's list of restricted addresses, then all traffic can flow to that address. This eliminates the need to have a constantly updated table of Internet sites in the Message Trusted Guard.

Whether or not a user can transmit to an address follows a simple rule:

```
if exists (recipientAddress)
then
   return (user not in recipientAddress.deny) and
          (user in recipientAddress.allow)
else
   return True    ;* address is not explicitly indicated - allow
endif
```

By default, the e-mail must go through. Therefore, it is up to the security officer to modify the transmission rights for particular sites. Although this is opposite to the typical "deny unless explicitly allowed" rule of computer security, it makes more sense for e-mail. For particularly sensitive destinations, the security officer should preconfigure the MTG to ensure some semblance of protection is provided.

In order to define those sites which are to be restricted and for which users, the following functions are defined:

removeSite *(site)*	*Remove* site *from the list of sites to which access is restricted.*
addSite *(site)*	*Add* site *to the list of sites to which access is restricted. Initially the allow and deny lists are empty.*
denySite *(user site)*	*Disallow* user *from transmitting to* site.
allowSite *(user site)*	*Allow* user *to transmit to* site.
removeUser *(user site)*	*Remove* user *from the allow or deny lists (the user can only reside on a single list).*

A Message Trusted Guard requires user identification and authentication. All users must be registered with the Message Trusted Guard prior to being allowed to transmit. If they are not in the Message Trusted Guard's list of users, all e-mail by the user is rejected and the action audited for future actioning by the security officer.

No trusted path is required by the Message Trusted Guard.

The Generic Policy Engine implicitly handles audit for the Message Trusted Guard. All mail messages are audited as the flow through the system by explicitly calling the GPE audit routines. Information stored includes sender, recipient, message ID, and status of the transmission.

Historically, the most difficult aspect of the Message Trusted Guard is all the supporting security software. Utilizing the GPE reduces the surrounding code to a mere fraction of C implementations. The implementation of the Message Trusted Guard Security Policy is defined below.

```
;* define "send" as "write" to make the security policy
;* that much more readable.
(define send 'write)
;*
;* The security policy proper
;*
(define (Message-Trusted-Guard sender messageID recipient)
   (cond
     ((not (entityExists? sender))
```

```
            (define status #F))                      ;;* deny send
        ((not (entityExists? recipient))
            (define status #T))                      ;;* allow send
        (else
            (let* (
                (Allowed (gpeAC recipient 'get send 'allow))
                (Denied  (gpeAC recipient 'get send 'deny)))
;           in
                ;* allow only if not explicitly denied,
                ;* but explicitly allowed!
                (define status (and (not (member sender Denied))
                                    (member sender Allowed)))))
      (gpeLog sender put (list sender messageID recipient status))
      ;* return the status of the send
      status
    )
```

Message Trusted Guard Security Policy Model in the GPE

The Message Trusted Guard Policy Model controls the transmission of electronic mail between a host site and select destinations. The following illustrates how a calling application can quickly deduce whether or not a message can be transmitted to the destination:

gpe (user@secret.org send dictator@banana.republic)

The function *gpe ()* invokes the security policy *Message-Trusted-Guard ()* and either a true is received granting transmission rights to the electronic mail message or a false is received, denying transmission rights.

The GPE offers a unique and flexible approach to creating security policies. The policies are defined solely as security policies performing a singular task: that of mediating information flow. The security policies implemented utilizing the GPE are short and easy to read guaranteeing fewer errors and omissions. Two example security policies have been provided illustrating the elegance of GPE implemented security policies. These security policies illustrate two popular security policies currently in use.

To illustrate the flexibility of the GPE, the following provides the code for the Biba Integrity Security Policy utilizing the existing code of the Bell-LaPadula Security Policy. As Biba is the complement of the Bell-LaPadula Security Policy it should be a simple matter to modify Bell-LaPadula into Biba. As can be seen, the only changes required (as noted) are trivial. Furthermore, the support routines for Bell-LaPadula remain unchanged.

```
(define (Biba subject access object)
   (let* (
      (Subject-Level (getLevel      subject))          ;;* MAC info
      (Subject-Cats  (getCategories subject))
      (Subject-Cavs  (getCaveats    subject))
      (Object-Level  (getLevel      object))
      (Object-Cats   (getCategories object))
      (Object-Cavs   (getCaveats    object))
      (Allowed       (getRight object access 'allow)) ;;* DAC info
      (Denied        (getRight object access 'deny)))
;  in
      (and (or (= Subject-Level Object-Level)          ;;* MAC
               ;;* no write up
               (and (> Subject-Level Object-Level)
                    (eq? access 'Write)                ;;* CHANGE
                    (subset? Object-Cavs Subject-Cavs))
               ;;* no read down
               (and (< Subject-Level Object-Level)
                    (eq? access 'Read)                 ;;* CHANGE
                    (subset? Object-Cavs Subject-Cavs)))

           (and (member subject Allowed)               ;;* DAC
                (not (member subject Denied)))

) ) )
```

Biba Security Policy Model in the GPE

By abstracting out the security to a GPE Server performance would be foremost on any implementor's mind. However, in terms of performance the GPE excels. The performance

of the *interpreted* GPE embedded within an HTTP Server on an Intel Pentium 166 running FreeBSD performance was clocked at 100,000 lines of HTML processed per minute through a simplified Bell-LaPadula Security Policy. There was no noticeable impact observed in the throughput of the server, even though each line had to be processed and approved by the security policy.

Currently the slowest aspect of the GPE is the entity space, which is managed akin to a simple database. Although this could be viewed as a possible trouble spot, there is no reason why a commercial-grade database couldn't be used to manage and maintain the entities on behalf of the GPE. The performance issue would disappear as would other, less obvious concerns regarding persistence and robustness of the current entity space implementation.

The current implementation of Idyllic is small and interpreted. It offers adequate performance in its interpreted state. However, being small allows it to be easily optimized and the use of a simple language based on Scheme allows for faster, compiled implementations.

As networks become more prevalent, distributed computing will become commonplace. Networks introduce a plethora of concerns not evident with stand-alone systems. Foremost among these is the need to ensure traffic is guaranteed to arrive at its destination without *any* modifications. The Generic Security Services API (GSSAPI) presented provides a baseline upon which to build. As users become more sophisticated and their security requirements increase in complexity, the GPE will provide system administrators with the ability to modify the security underpinnings to meet user demands.

We claim:

1. A method of controlling access to a network wherein security policies are defined by using a verifiable language consisting of formal definitions of the syntax and semantics, whereby security policy is abstracted from physical data.

2. A method as claimed in claim 1, which is controlled by a generic policy engine (GPE) , which executes the policies defined by said language.

3. A method as claimed in claim 1, wherein said GPE mediates access to an object entity by a subject entity, there being a series of defined actions that can be performed on each entity.

4. A method as claimed in claim 2, wherein said GPE encapsulates security information in entities describing attributes, policy and relationships with items to be protected.

5. A method as claimed in claim 4, wherein said entities are mapped on a one-for-one basis with items to be protected.

6. A method as claimed in claim 5, wherein each said entity has a unique identifier.

7. A method as claimed in claim 6, wherein said GPE provides Application Programming Interfaces (APIs) to permit third party applications to manipulate security attributes and thereby determine information flow.

8. A method as claimed in claim 7, wherein said APIs comprise an entity API, a Security Policy API, an identification and authentication API, an audit API, and Access Rights API, and a Privileges API.

9. A method as claimed in claim 7, wherein each calling application obtains its own instantiation of the GPE.

10. A security policy engine for controlling access to a network in accordance with policies defined with the aid of a verifiable language consisting of formal definitions of the syntax and semantics.

11. A security policy engine as claimed in claim 10, comprising a plurality of Application Programming Interfaces to receive requests for access controls from third party applications.

12. A security policy engine as claimed in claim 10, comprising entities containing security information for external items.

13. A security policy engine as claimed in claim 10, further comprising language support libraries for said verifiable language.

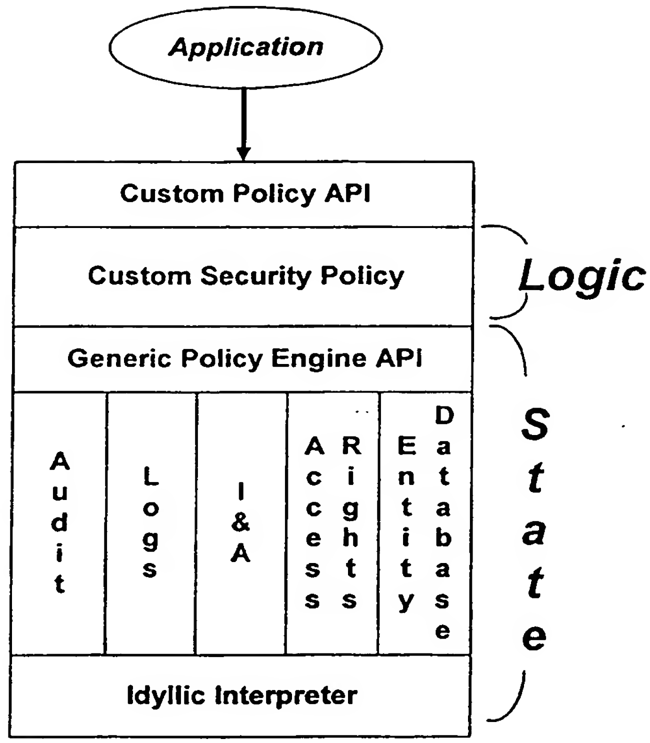

Application
Custom Policy API
Custom Security Policy
Logic
Generic Policy Engine API
Audit
Logs
I&A
Access
Rights
Entity
Database
State
Idyllic Interpreter

Fig. 1


Subject
Object

Fig. 3

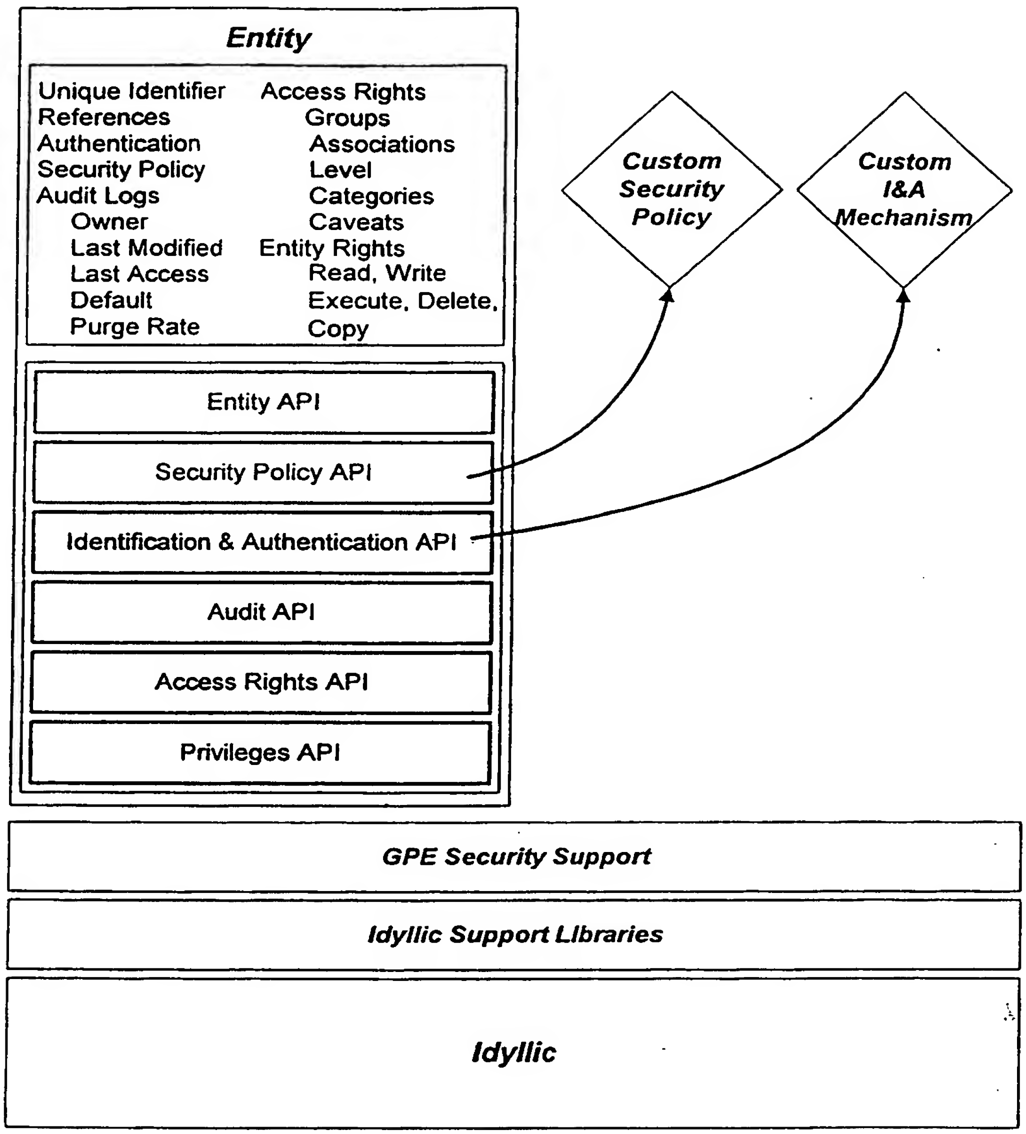

Entity
Unique Identifier
References
Authentication
Security Policy
Audit Logs
Owner
Last Modified
Last Access
Default
Purge Rate
Access Rights
Groups
Associations
Level
Categories
Caveats
Entity Rights
Read, Write
Execute, Delete,
Copy
Entity API
Security Policy API
Identification & Authentication API
Audit API
Access Rights API
Privileges API
Custom Security Policy
Custom I&A Mechanism
GPE Security Support
Idyllic Support Libraries
Idyllic

Fig. 2

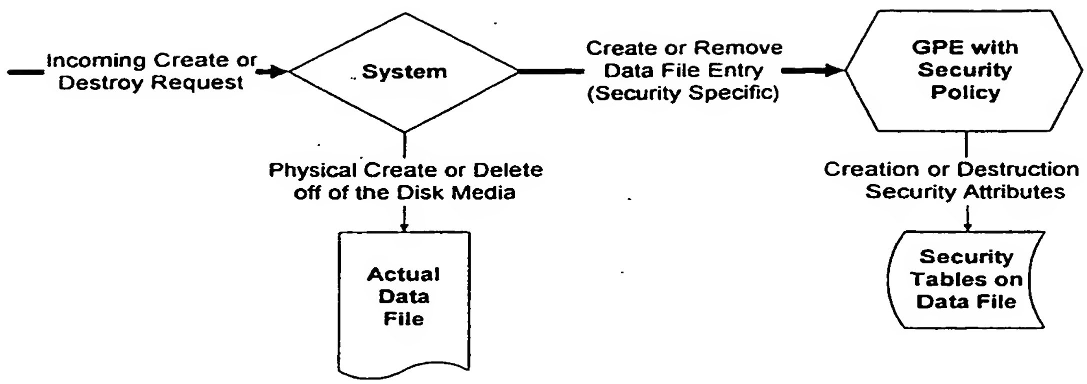

Incoming Create or Destroy Request
System
Create or Remove Data File Entry (Security Specific)
GPE with Security Policy
Physical Create or Delete off of the Disk Media
Creation or Destruction of Security Attributes
Actual Data File
Security Tables on Data File

Fig. 4

INTERNATIONAL SEARCH REPORT

International Application No
PCT/CA 00/00276

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 H04L29/06

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 H04L

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal, WPI Data, PAJ, IBM-TDB

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	VARADHARAJAN, V.; CRALL, C; PATO, J.: "Issues in the design of secure authorization service for distributed applications" GLOBAL TELECOMMUNICATIONS CONFERENCE, 1998. GLOBECOM 1998. THE BRIDGE TO GLOBAL INTEGRATION. IEEE, 'Online! vol. 2, 8 - 12 November 1998, pages 874-879, XP002142413 ISBN: 0-7803-4984-9 Retrieved from the Internet: <URL:www.iel.ihs> 'retrieved on 2000-07-12! abstract page 874, right-hand column, line 39 -page 875, left-hand column, line 11 page 875, right-hand column, line 1 -page 876, left-hand column, line 19 page 877, left-hand column, line 25 - line -/--	1-13

[X] Further documents are listed in the continuation of box C.

[ ] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
12 July 2000	26/07/2000

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Adkhis, F

1

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
	44 page 878, right-hand column, line 6 -page 879, left-hand column, line 32	
A	CHANG, S.K.; POLESE, G.; THOMAS, R.; DAS, S.: "A Visual language for authorization modeling" VISUAL LANGUAGES, 1997. PROCEEDINGS. 1997 IEEE SYMPOSIUM ON, 'Online! 23 - 26 September 1997, pages 110-118, XP002142414 ISBN: 0-8186-8144-6 Retrieved from the Internet: <URL:www.iel.ihs> 'retrieved on 2000-07-12! abstract page 110, right-hand column, line 28 -page 111, left-hand column, line 24 page 115, left-hand column, line 27 -right-hand column, line 51	1-13

1